IEEE MICRO

JUNE 1988

Embedded, 32-bit processors enrich both ordinary and exotic applications

THE COMPUTER SOCIETY

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

Okay, okay! We agree that the reliquary of Emperor Charlemagne doesn't really have microprocessors amidst its other embedded jewels. But, isn't it stunning to see this medieval museum piece and our modern products combined in such a dramatic fashion?

We thank Thames & Hudson Ltd. in London for letting Charlemagne adorn the cover.

Cover design: Jay Simpson

Departments

Change-of-Address form, p. 2; Classified Ads, p. 104; Reader Service Cards, p. 104A

IEEE MICRO

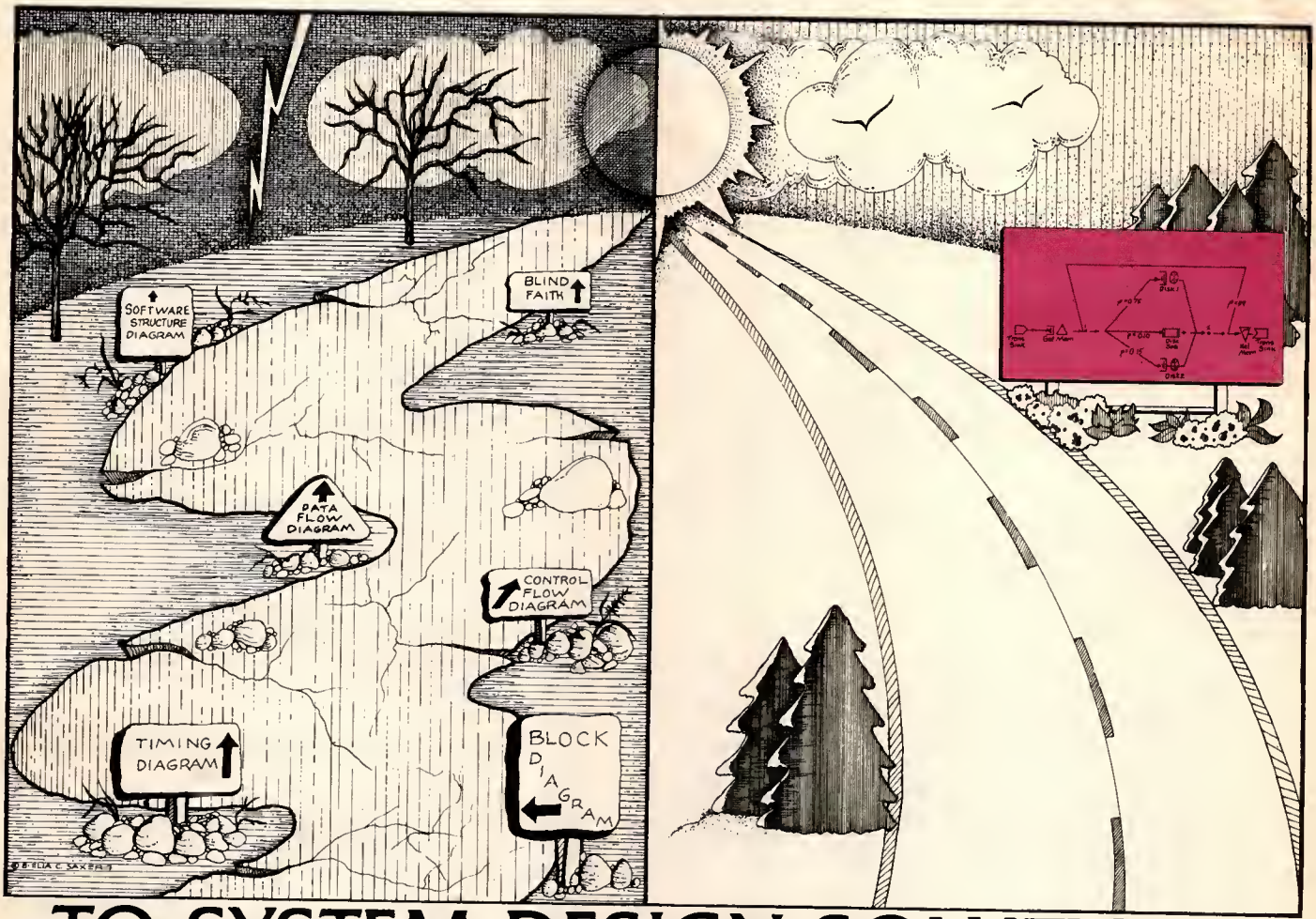
Volume 8 Number 3 (ISSN 0272-1732) June 1988

Published by the Computer Society

Feature Articles

Special Feature

GET ON THE "PAWS™ EXPRESSWAY"



TO SYSTEM DESIGN SOLUTIONS

Why not take the shortest and fastest route toward system design solutions? To define and model a large, complex system, you need a modeling tool that can address the system's total architecture.

PAWS/GPSM™ is an easy-to-use high productivity modeling and simulation tool that lets you evaluate alternative architectures while focusing on performance. It provides a state-of-the-art environment for developing accurate and reliable product definitions.

PAWS provides high level primitives closely resembling the user's intuition. PAWS models the behavior of a total system implemented in diverse technologies including software, hardware, and firmware.

GPSM, the graphical interface to PAWS, helps you quickly and easily transfer your ideas into pictures. GPSM helps you visualize multiple data and control flows through several components and incrementally synthesize models of a total system.

PAWS/GPSM is flexible. It lets you refine your model as your system design evolves so you can eliminate potential problems as early as possible in the product design cycle.

Call (512) 474-4526 to receive information on PAWS/GPSM. And get on the PAWS expressway to system design solutions.

PAWS and GPSM are trademarks of Information Research Associates, Inc.

IRA

Information Research Associates • 911 W. 29th St. • Austin, Texas 78705 • **(512) 474-4526**

Reader Service Number 1

Moving?

Address changes:
Please notify us 4 weeks in advance

Name (Please Print)

New Address

City

State/Country Zip

ATTACH LABEL HERE

- This address change notice will apply to all IEEE publications to which you subscribe.
- List new address above.
- If you have a question about your subscription, place label here and clip this form to your letter.

Mail to:
IEEE Micro, Circulation Dept., 10662 Los Vaqueros Cir., Los Alamitos, CA 90720-2578

Circulation: *IEEE Micro* (ISSN 0272-1732) is published bimonthly by the Computer Society of the IEEE: IEEE Headquarters, 345 East 47th St., New York, NY 10017; Computer Society West Coast Office, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. Annual subscription: $18 in addition to Computer Society or any other IEEE society member dues; $27 for members of other technical organizations. Single copies: $10 for members; $20 for nonmembers. This journal is also available in microfiche form.

Postmaster: Send address changes and undelivered copies to *IEEE Micro,* 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. Second class postage is paid at New York, NY, and at additional mailing offices.

Editorial: Unless otherwise stated, bylined articles and descriptions of products and services in New Products, Product Summary, MicroReview, MicroNews, MicroView, and MicroLaw, reflect the author's or firm's opinion; inclusion in this publication does not necessarily constitute endorsement by the IEEE or the Computer Society of the IEEE. Send editorial correspondence to *IEEE Micro,* 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. All submissions are subject to editing for style, clarity, and space considerations. *IEEE Micro* subscribes to The Computer Press Association's code of professional ethics.

Department

From the Editor-in-Chief

The Adams Report

During 1987, at the commission of the Computer Society, The Adams Company, a Connecticut media and marketing research firm, performed a subscriber profile for *IEEE Micro*. Conducting this survey periodically helps us ascertain who our readers are and what direction the magazine should take or continue. Survey responses also allow us to more carefully direct our efforts to increase our subscriber base and find advertising.

Adams mailed a lengthy survey questionnaire to 1,000 subscribers along with a US dollar bill as a small prepaid "reward." Adams also sent a follow-up letter two weeks later. An amazing 568 recipients responded (457 from the US, 111 from all other nations). The resultant subscriber study, including reader comments, charts, and conclusions, fills over 200 pages. Since the volume of the responses constitutes a statistically significant sample, we can extrapolate the results to the entire population of *IEEE Micro* readers. I summarize some of the Adams report's interesting findings for you.

First, you are a pretty highly educated group:

- over 94 percent of you possess at least a bachelor's degree,
- nearly 54 percent secured a master's degree, and
- 17 percent earned a PhD degree.

This is indeed impressive.

Unsurprisingly, practically all of you have access to a computer:

- 98 percent have a computer at home or at work,
- 84 percent are computer literate,
- 75 percent use PCs, and
- 42 percent use Apples (mostly Macs).

Obviously, many of you use both systems—an unexpected result.

In the words of The Adams Company, "*IEEE Micro* delivers (to the advertiser) a highly qualified audience of systems and software designers, engineers, and sophisticated end users of computer systems and equipment." Further, at your companies,

- 72 percent of you select or recommend computer systems or hardware, and
- 73 percent select or recommend software tools or programs.

A high percentage of you take part in purchasing hardware or software:

- 77 percent with the purchase of computer systems, CPUs, or peripherals;
- 72 percent with the purchase of software;
- 49 percent with software development or software tools; and
- 24 percent with integrated circuit or microprocessor development.

It is my hope that our advertising representatives can use these findings to convince some potential advertisers to buy space. We can use the money, page color, and informational value to readers that advertising brings.

Practically all of you affect some aspect of computers or integrated circuits:

- 86 percent of you work with research, design, or development of computers or electronic components, systems, or software for your company;
- 24 percent are software engineers or scientists (a surprisingly high percentage);
- 18 percent are hardware engineers or designers;
- 17 percent are system developers; and
- 10 percent are managers in the computer field.

"*IEEE Micro* is well read by its subscribers," says Adams. It concludes from the responses that an average reader spends one hour and 21 minutes on each issue. I feel very good about this finding. Very busy engineers usually read material very quickly, never dawdling. Responses also indicated that

- 63 percent of you saved an article or an ad (very few saved ads, I am sure); and
- each copy is read by 2.2 readers, on average.

Most of you read other technical publications as well:

- 85 percent read *Computer*,
- 37 percent read *IEEE Software*,
- 27 percent read *Byte*,
- 20 percent read *EDN*,
- 13 percent read *Electronic Engineering Times*,
- 13 percent read *Electronics*,
- 11 percent read *IEEE Spectrum*,
- 10 percent read *Computer Design*,
- 9 percent read *Communications of the ACM*,
- 9 percent read *Electronic Design*, and
- less than 9 percent read some 14 other magazines.

When asked what features were found to be the most interesting or useful, readers offered answers that ranged over the entire magazine. As expected, the technical articles, focusing on new developments and tutorials, led the way.

The mailbag:

"...more on LAN and data acquisition." R.M., Rosario, Argentina

"I liked MicroStandards. Where can I get some of the standards mentioned?" A.R.B., Mexico City (A.R.B., a faithful reader, writes frequently. I will forward this request to Michael Smolin, our MicroStandards editor.—J.F.)

"Well-illustrated and easy-to-understand articles.... Comparison with VAX is incorrect because VAX performance was done under non-native Unix 4.3...need more information about instruction set of transputer family and details of *IEEE Standard 754-1985*." I.S., Moscow, USSR (An IEEE publications catalog will be sent to you as you requested. For more details on the transputer instruction set, I suggest that you contact Inmos in the United Kingdom.—J.F.)

"...more on parallel processing and RISC." J.K., Mulgrave, Australia

"...more on expert systems and artificial intelligence." B.G.Y., Inchon, Korea

"...more on math processors and RISC computers." A.M., Tehran, Iran

"...more on neuronlike computing." E.D'M., Ghent, Belgium

"...liked MicroLaw and VMEbus software...would like an article on 68030..." M.L., Tienen, Belgium

"Article on GaAs was great—useful, timely, in-depth, clear, fantastic. Video RAM article very insightful. MicroLaw was great as usual. Disliked Balance microprocessor article—lacked something—all package, no beef!" T.T., San Pedro, CA

"...more on transputers." R.G., Chicago

"I liked 'Dirty Harry' (MicroLaw). Keep up the good work!" B.M., Lynchburg, VA

"...more on PS/2." M.H.T., Chicago

"...liked component-level hardware articles and New Products (optical and handicapped)." K.A.C., Ottawa, Ontario

"...would like to see CISC, RISC, SISC, WISC, TRON (all types) together in one article describing their respective relationship." R.L.Z., Munich (This would prove to be a very interesting comparison indeed! I would be very interested in reviewing such an article for publication. Any takers? —J.F.)

"I liked MicroView on WISC." S-H.W., Groton, CN (A good job by Christine Miller, our assistant editor.—J.F.)

"Excellent work...more articles." M.A.S., Ottawa

"I disliked Milutinovic's article." A.G., Urbana, IL

Readers also reported great interest in news, new products, law, and product reviews. Microprocessors (all kinds) and MicroLaw received a very large number of mentions.

Now the questions and the answers become more difficult. Question 18 asked, "What would you like to see more of in *IEEE Micro*? Adams reports well over 200 answers to this question. I read them all and will submit them to the entire *IEEE Micro* Editorial Board for study prior to our annual meeting. However, let me share with you my impressions of the comments.

The fundamental aspects of *IEEE Micro* appear to be pretty much on track. However, many readers wish a greater number of technical articles about very recent developments on DSPs, RISCs, and other MPUs. Many readers would like to see comparative articles, applications, and tutorials on new technology. Several also mentioned specific application aspects of computer and microprocessor technology. Several readers disliked theoretical articles, or those that had little "real-world" application.

Personally (and this feeling is backed by past reader correspondence), I am concerned when articles run over 15 magazine pages. We certainly want to maintain the in-depth character of our technical articles, but we must be cognizant of our page count and attention-span constraints. Our readers are all busy people.

Jim Farrell

Letters to the Editor

Afterthoughts bring response

To the Editor:

After reading the introduction to the February MicroReview, I looked back at all Richard Mateosian's previous MicroReviews. I found that I had already read them all—and enjoyed them as well.

Keep writing about books, please, Mr. Mateosian. And, yes, I would like to see opinions on desktop publishing software for IBMs or MACs. You *do* have readers!

Luc Faubert
Montreal, Quebec

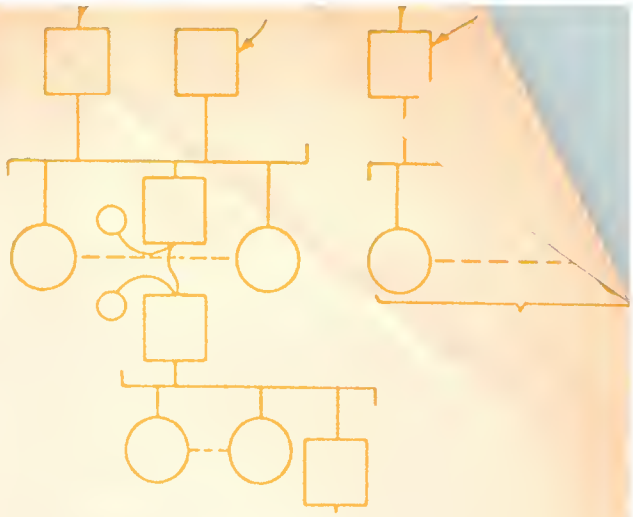
Notice to subscribers

Much to our chagrin, we note that some April 1988 issues of *IEEE Micro* were incomplete. If you received an issue missing several pages, we will be happy to send you a replacement copy. Address your request to:

IEEE Micro
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
Attn: Gaye Seaborn

Department

MicroStandards



Michael Smolin
Smolin & Associates
3428 Greer Road
Palo Alto, CA 94303

IEEE 32-bit, backplane bus comparisons

IEEE Micro has occasionally carried tables comparing the salient features of backplane buses that qualify as either IEEE standards or standards drafts. The table on the following page addresses buses that are IEEE standards (or in the case of P1096, a final proposed draft). This table compares six 32-bit backplane buses that use two-piece mating connectors and is intended for handy feature reference. I drew it in part from a larger work developed by James (Bob) Davis and presented in part as a tutorial at the Anaheim Buscon. Fastbus, Futurebus, VMEbus, VSB, Nubus, and Multibus II make up the list.

Fastbus (*IEEE Standard 960, Modular High-Speed Data Acquisition and Control System*) is in use at over 12 sites (about 1000 systems), including SLAC at Stanford and Switzerland's CERN (European Organization for Nuclear Research). It has the highest speed and lowest cost of the full-featured buses compared here.

Futurebus is the newest of these buses. It is intended to be an asynchronous bus supporting block transfer, broadcast, and transfer intervention. Some Futurebus specifications have been adopted as IEEE Standard 896.1, Futurebus: A Backplane Bus Specification for Multiprocessor Architectures. (I expect publication in three to six months.) Additional Futurebus specifications are currently under development in the P896.2 working group. These specifications will standardize module identification, interrupt mechanisms, a message-passing format, and a general-purpose cache coherence protocol. The working group chair expects completion of P896.2 by November.

VMEbus is ubiquitous in today's market with its thousands of installations and hundreds of manufacturers. It uses subbuses as an inherent part of its architectural design. VMEbus architectural concepts are based on the Versabus, which was developed in the late 1970's to support the 68000 microprocessor. VMEbus (IEEE Standard 1014, Versatile Backplane Bus) was introduced to the market in 1980 and is the oldest bus compared here.

VSB (Proposed IEEE Standard 1096, Multiplexed High-Performance Bus Structure) was designed to be the VMEbus subsystem bus. It off-loads the global VMEbus backplane by providing its own high-performance access to memory and I/O. It is also eminently suitable for use as a high-performance system backplane independent of VMEbus.

Of the six IEEE buses in this comparison, only Nubus and Multibus II are synchronous buses. *Nubus*, a high-performance backplane bus, represents a distinctly minimalist approach for multiprocessor systems. Officially it is IEEE Standard 1196, Nubus: A Simple 32-Bit Backplane Bus.

Multibus II (IEEE Standard 1296, A Full-Feature 32-Bit Backplane Bus) is a full-featured, high-performance, synchronous multiprocessing bus that supports message passing.

Bus experts have checked the data contained in Table 1 for accuracy. While I believe in its accuracy, the usual admonitions and disclaimers apply—let the user be wary. Direct any questions to me or to Bob Davis, Summit Consulting, 22685 Summit Road, Los Gatos, CA 95030, (408) 353-2706. Davis, an independent consultant, is the former chair of both the Microprocessor Standards Committee and the 896 Futurebus working group.

Some salient thoughts

As with any comparisons, benchmarks, and so on, the choice of comparison parameters and methods for quantifying them can slant the results. We have tried to be comprehensive in the parameters presented so that users may intelligently choose which bus might best solve their particular problems.

Further, just because a bus *meets* standards specifications does not guarantee that the bus system will *function* properly. To achieve optimal performance, a bus profits from generally undocumented, "good engineering design practices." These practices extend, for example, to the very geometric arrangement of boards installed in the backplane—the more uniform distribution of a large number of boards the better. (For very small systems, it is wise to keep the boards adjacent, with the clock source in the center, or in slot 1 as required for Nubus or Multibus II.)

Also, some backplane buses do not have adequate grounds. Remember that every signal current must complete a circuit loop. The switching transient current sees an effective impedance of about 19 or 20 ohms. When 32 signal lines are driven simultaneously, the ground level can bounce high enough for erroneous 1's to be detected. In many systems, up to 80 lines may be switched simultaneously on the primary bus (the switching of one or more secondary buses aggravates the problem). Delaying the signal-latching strobes to allow transients to die out resolves this problem at the sacrifice of speed. The secondary bus, running asynchronously to the primary bus (such as VSB in a VMEbus system) can add so many other switching transients that even this solution becomes inadequate. The respective standards have not considered these potential failure modes. Although these delays are significant, I have never seen them factored into bus performance calculations.

Remember that the IEEE makes no comments about product compliance with any of its standards. Indeed, even if several products did comply with a standard, the possibility always exists that they will not work together. Many features in each standard are allowed, but not required. If your system initially implements any of these allowed functions, then you must be careful that any add-on boards maintain those features. In a future issue I will try to present some of those features that are not required per a standard, but whose omission may cause system unre-

Table 1. Comparison of 32-bit IEEE standard backplane buses.

Features	IEEE Standard 960	896.1	1014	P1096	1196	1296
Common name	Fastbus	Futurebus	VMEbus	VSB	Nubus	Multibus II
License fee	No	No	No	No	Yes	Yes
System cost	Low	Medium	Low/medium	Low/medium	Low/medium	Medium
Syn./asy.	Asy.	Asy.	Asy.	Asy.	Syn.	Syn.
Multiplexed	Yes	Yes	No	Yes	Yes	Yes
Board size (in.)[1]	14.4 × 15.7	11 × 14.4	6.3 × 9.2	6.3 × 9.2	4 × 13	9.2 × 8.7
Address spaces	2	1	64	4	1	4
Address widths (bits)	32+[2]	32	16/24/32	32	32	32
Data xfer widths (bits)	32	8/16/24/32	8/16/24/32	8/16/24/32	8/16/32	8/16/24/32
Address/data parity	Yes	Yes	No	No	Yes	Yes
Memory space	4GW	4G – 32M[3]	64K to 4G	4G	4G – 256M[4]	4G
Dedicated space						
I/O	4GW	32M[3]	No[5]	4G	256M[4]	64K
Message	No	No	No	4G	No	255/255 bytes[6]
CSR	4GW	32M[3]	No[5]	4G	256M[4]	No
Interconnect	No	No	No	No	No	512 bytes/slot
Transfer cycle types	R/W	R/W	R/W/V	R/W/A/V	R/W	R/W/Msg.
Bus handshake	Yes	Yes	Yes	Yes	Yes	Yes
Multimaster	Yes	Yes	Yes	Yes	Yes	Yes
Arbitration	Par	Par	DC	DC/Par	Par	Par
Fairness	Yes	Yes	Yes	Yes	Yes	Yes
Priority	Yes	Yes	Yes	Yes	No	Yes
Parity	Yes	Yes	N/A	Yes	No	No
Event cycles	Yes	Yes	—[7]	Yes	Yes	Yes
Interrupt type	Event	Event	7 line[7]	1 line, event	1 line, event	Event
Interrupt ack.	Yes	No	Yes	Yes	No	Packet ack.
Geographic add.	Yes	Yes	No	Yes	Yes	Yes
Message protocol	No	No	No	No	No	Yes
Block transfer	Yes	Yes	Yes	Yes	Yes	Yes
Broadcast	Yes	Yes	No	Yes	No	Msg.
Broadcall	Yes	Yes	No	No	No	No
No. of slots	26-32	21-32	21	8	16	21
No. of backplane layers	7	>5	6	4-6	>6	8
Central resources required	Yes	No	Yes	For DC	Clock	Yes
Connector type	Modu	DIN	DIN	DIN	DIN	DIN
Performance (/second)[8]	160-200M	60-100M	30M	30-60M	37.5M	32-40M

1. Board dimensions represent approximate dominant sizes.
2. Fastbus addresses words, not bytes. Each 32-bit address selects a module; the following 32-bit address selects a word in that module, giving an effective 64-bit handshake.
3. Futurebus reserves 32M bytes of its 4G-byte memory space for slot memory (32 × 1M bytes = 32M bytes). Some of this slot memory may be used for I/O space, some for CSR space.
4. Nubus reserves 256M bytes of its 4G-byte memory space for slot memory (16 × 16M bytes = 256M bytes). Some of this slot memory may be used as I/O space, some for CSR space.
5. Although the VMEbus standard does not define an I/O or CSR space, these spaces can be made from one of the 64 memory spaces.
6. Bidirectional, 255 bytes each way.
7. VMEbus supports an address-only cycle without handshake. An event cycle requires handshake.
8. Maximum claimed transfer speed for two adjacent available boards. Probably the most controversial column in this table. A company wants its product to look good. I insist that the performance given be achievable with products available today. A speed range reflects either variations across manufacturers or arguments over definitions.

Terms
A = Arbitration
DC = Arbitration by daisy-chain
G = Gigabyte
GW = Gigaword
M = Megabyte
Msg. = Message
Par = Parallel arbitration scheme
R = Read
V = Vector
W = Write

Each memory space size is given in terms of the number of addressable locations.
4G-32M means that 32 M bytes of the 4G-byte space is reserved for another function that is usually given below.
Dedicated means defined by requirements of the specifications in a standard, and not that an implementer can find an innovative (or other) way to perform the function without violating the standard.

liability. Be careful when you construct your compatibility/compliance matrix.

International activity

At its recent 1988 meeting, SC47B accepted a US proposal to process six IEEE standards for international status. SC47B, a committee under Joint Technical Committee 1 (JTC1), was formerly a subcommittee within the IEC (International Electrotechnical Commission). JTC1 is the new joint IEC/ISO (International Organization for Standardization) committee with responsibility for standards in information technology.

The six IEEE standards are:

- *IEEE Standard 854, Radix-Independent Floating-Point Arithmetic,*
- IEEE Standard 896.1, Futurebus: A Backplane Bus Specification for Multiprocessor Architectures,
- IEEE Standard 961, An 8-Bit Microcomputer Bus System (STD),
- IEEE Standard 1000, STE Bus: Specifications for Standard Eurocard 8-Bit Backplane Interface,
- IEEE Standard 1196, Nubus: A Simple 32-Bit Backplane Bus, and
- IEEE Standard 1296, A Full-Feature 32-Bit Backplane Bus.

IEEE Standard 1101, Mechanical Core Specifications for Microcomputers is to be incorporated into 896.1, 1196, and 1296 as the final chapter of each standard. In addition, SC47B is sending IEEE 1101 to Technical Committee 48 with the recommendation that IEEE 1101 be adopted as an ISO/IEC standard.

The six proposals will be balloted internationally under JTC1 rules after the IEEE completes publication of these standards.

SC47B also considered the adoption of an international standard for assembly language mnemonics. IEEE 694 and Calm, two opposing proposals for such a standard, had already been rejected by SC47B. The committee also rejected a hybrid proposal of these two by a close floor vote (five to four). The committee later reopened and approved the subject for a mail ballot of member national bodies. If you have an interest in this subject, drop me a note requesting a copy of the document. I will expect your feedback in return.

Other news

The TCMM has submitted a Project Authorization Request (PAR) to the IEEE Standards Board for approval of a project to develop standards for Scheme, a Lisp-like computer language. I expect that more PARs will be submitted in August for projects on Superbus and database algorithmic definitions. [*See related Superbus item in MicroNews this issue.—Ed.*]

The TCMM has submitted two proposed standards to the IEEE Standards Board for formal adoption. They are P959 I/O Extension Bus (SBX), and P1096 Multiplexed High-Performance Bus Structure (VSB). P970 Advanced Backplane Bus (Versabus) is expected to follow suit in a couple of months.

Official IEEE procedures require reaffirmation, revision, or withdrawal of an IEEE standard every five years, if not sooner. Consequently, the TCMM is submitting *IEEE 796, A Standard for a Microcomputer System Bus* (Multibus I) to the IEEE Standards Board for reaffirmation.

Call for sponsor ballot participation

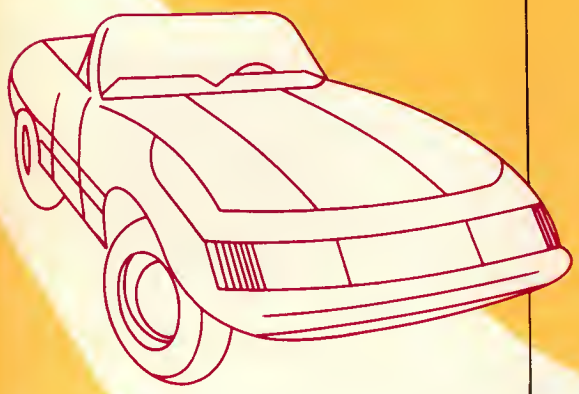
I expect that a revision of *IEEE 855, MOSI—A Standard for Microcomputer Operating System Interfaces* will be ready for its sponsor consensus ballot in a month or so. This ballot will recommend adoption of the revised standard to the IEEE Standards Board. Also, a balloting body for upgrading *IEEE Standard 695, Trial-Use Standard for Microprocessor Universal Format for Object Modules* (MUFOM) to full-standard status is now being formed. I expect to form balloting bodies for P896.2 and P1196.2 later this year. Please let me know if you want to participate in any or all of these sponsor ballots for recommending the adoption of each draft to the IEEE Standards Board. (Include your IEEE or Computer Society membership number.)

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 174 **Medium** 175 **High** 176

Another Cost-Effective Design Tool



Jack Grimes
Intel Corporation

Joe Hootman
University of North Dakota

This special issue of *IEEE Micro* emphasizes the concept of the embedded processor. In particular we present discussions of the high-end embedded processors that contain 32-bit programming models. These processors support high-performance applications such as local area network servers and 20-page/minute laser printers.

Most users don't see embedded processors. In fact, these processors often control a product not normally considered to be a computer. Designers program the highly integrated, commodity price-driven processor/controller to fit a particular range of applications. If there is a universally accepted definition of an embedded processor, we have not seen it. The best definition comes from looking at examples of how these devices are used.

Embedded processor applications include microwave ovens, laser printers, programmable scales, robots, navigation systems, and automobile antiskid braking systems, engine control, adaptive suspension systems, and audio systems. For years the computer industry has used embedded processors in smart terminals, disk controllers, and communications interfaces.

These processors can be the same general-purpose devices found in your favorite personal computer. However, demand is growing for low-cost processors specifically designed for embedded applications. These devices are, as a rule, variations of existing processors with some functions modified to reduce cost yet still allow the designer maximum functionality for a particular range of applications. In general, embedded applications tend to make use of smaller amounts of memory and make extensive use of EPROMs for program storage. Some embedded applications, such as laser printers and plotters, can access over 2M bytes of memory.

Dataquest, a respected market research firm, estimates that in 1987 over five embedded controllers were sold for every microprocessor sold. That estimate translates to 90 million microprocessors and about 500 million embedded processors sold last year. Obviously, this area of technology cannot be overlooked by the industry or by the practicing design engineer.

In this issue we offer technical and explanatory information about five processors, which we consider to be embedded processors: the 80376, VL86C010, TMS34010, R3010, and 80960. Intel authors discuss how the 80376 was derived from the 80386 in such a way that the user can depend on existing software development tools. They also discuss a companion multi-function chip and give several application examples.

072-1732/88/0600-0008$01.00 © 1988 IEEE

Texas Instruments originally designed the 34010 as an embedded graphics processor. The 34010 designers used general-purpose RISC ideas and added hardware modules to supply DRAM refreshing and bitmap graphics.

VLSI Technology's author recounts the development of a new embedded controller, the VL86C010, which works in an application environment that is interrupt and I/O intensive.

MIPS Computer Systems presents its newly developed RISC floating-point coprocessor, the R3010. This processor is remarkable because of its impressive performance figures.

We round out the issue with a second Intel article on the new architecture of the 80960 embedded processor, which efficiently executes code. The high-performance 80960 incorporates many RISC processor ideas and includes an on-chip floating-point unit.

The future for embedded processors looks extremely bright. We anticipate that future design engineers will consider these processors to be one of their fundamental design building blocks. At the present time embedded processors support a variety of applications that would not be possible—or at least would be extremely difficult—to carry out without their use.

The addition of this improved technology impacts the design process and the way that design engineers think about their designs. With the embedded processor the design engineer gains another cost-effective tool for efficient system design.

Jack Grimes is a graphics system architect with Intel Corporation in Santa Clara, California. He has also been a research director at the ITT Advanced Technology Center in Stratford, Connecticut. While there, he held responsibility for an applied research group working on office automation for software development in large projects. His current research interests include user interfaces, concurrent systems architecture, and management science.

Grimes holds BS, MS, and PhD degrees from Iowa State University in electrical engineering and computer science and an MS in experimental psychology from the University of Oregon. He has served as technical editor of *Computer* and on the editorial boards of *ACM Computer Surveys, IEEE Software,* and *IEEE Computer Graphics and Applications.* From 1980 to 1988 he organized and chaired tutorials at SIGGRAPH on human factors and user interface design.

Joe Hootman is professor of electrical engineering at the University of North Dakota, where he currently holds responsibility for the School of Engineering and Mines CAD Laboratory. He has participated in many National Science Foundation and American Society of Engineering Education faculty development programs. He also served as the technical sessions coordinator for the national ASEE conference held on the UND campus. Before assuming his present position, he was associate professor at Colorado State University and worked with the Environmental Science Services Administration in Boulder, Colorado.

Hootman received his BSEE from the University of Missouri at Rolla and his MS and PhD from Iowa State University. He is a member of ASEE, ACM, IEEE, Tau Beta Pi Sigma Xi, and Eta Kappa Nu. For the past three years he has served as Associate Editor-in-Chief of *IEEE Micro.*

Questions concerning this special issue can be addressed to either guest editor. Contact Grimes at Intel Corporation, SC4-40, 2625 Walsh Avenue, PO Box 58122, Santa Clara, CA 95052-8122. Hootman's address is Electrical Engineering Department, University of North Dakota, PO Box 7165, Grand Forks, ND 58202.

FEATURE

The Intel 376 Family for Embedded Processor Applications

You can build a 3-MIPS embedded system on a 4 × 5-inch board with this 32-bit family. It eliminates 30 LSI/VLSI components needed in other 32-bit systems.

Clif Purkiser and Jim Kardach

Intel Corporation

Today's 32-bit microprocessors fit into the systems of a vast array of products ranging from reprogrammable applications, such as engineering workstations and personal computers, to embedded applications, such as digital switching networks, robotics, and laser printers. Intel specifically designed its 80376 family to meet the needs of the embedded application marketplace. Here, we look in detail at the register set, addressing modes, instruction set, protection mechanisms, and debugging features of the 376 processor, especially as they relate to embedded applications. We also look at three other members of the family: the 82370 integrated system peripheral; the 27C203A pipelined, 256-kilobyte EPROM; and the 80387SX numeric coprocessor.

The embedded application market

Embedded processors, or "hidden computers," control everything from automatic coffee makers to very sophisticated telecommunications network systems. These preprogrammed processors execute specific functions. The 32-bit microprocessor works particularly well in communications control, industrial process controllers, and robotics. Specific product examples are avionics/radar central office switches, private automatic branch exchanges (PABXs), local area networks, network servers (managing many functions on LANs), sophisticated robots, and cluster controllers (managing robot groups). All of these embedded system applications demand real-time response that lets systems deal with events as they arise, with no significant delay.

Embedded applications have several identifiable needs. Ideally, components used in 32-bit embedded applications should lead to systems that are low in overall cost. Components must be physically small and capable of tolerating a wide variety of environments, including harsh ones. For input/output functionality, embedded applications typically require direct memory access (DMA), wait-state control, interrupt control, and timing facilities. The best components for the embedded marketplace provide an easy-to-design system with low development and manufacturing costs. They also offer gains in the speed of computation and increase the range of functions that embedded systems can perform.

072-1732/88/0600-0010$01.00 © 1988 IEEE

Customizing for embedded applications

Designers based the 376 embedded processor on the 386 processor architecture it complements and then customized it for embedded applications. As a result it supports features needed by embedded applications and is compatible with the 386 processor's 32-bit mode. But it does not support features needed exclusively by personal computers and workstations. For example:

- The 376 processor is not backwardly compatible with the 8086 or 80286 processors because binary compatibility is not a critical factor for most embedded applications. Very rarely do designers upgrade an embedded application by simply replacing the existing CPU with a more powerful one and using the old EPROMs (erasable programmable ROMs). Instead, the migration of embedded applications from one generation to the next typically involves a significant change in hardware, which in turn mandates software modifications.
- Since most embedded applications do not need paging, the 376 processor does not support it. The native, 32-bit 386 processor environment known to programmers as protected mode permits all 376 programs to be executed on 80386 machines. But the 376 processor executes only 386 processor 32-bit programs that are not dependent on paging.

Crucial to system design is the availability of quality development tools. These tools enhance system performance and provide designers with software and debugging products to ease the time-consuming tasks associated with microprocessor design. They need to be easy to use yet flexible enough to meet the diverse needs of embedded applications. Since the 80376 is based on existing 80386 architecture, the embedded application design engineer using the 376 processor can call on:

1) A broad base of compatible development tools that already exist. Designers can use 80386 tools including those on different operating systems (DOS, Unix, VAX/VMS) and hosts (Sun, Apollo).

2) A choice of real-time, IBM PC AT, 80386-based operating systems that are available for prototyping applications. Designers can develop and test their applications on existing personal computers. Applications that run in this environment also run on the 376 processor.

3) A capability for using existing 386 processor-specific, real-time kernels such as the iRMK.

4) The availability of 32-bit language translators for software development. These include ASM-386, C, Pascal, Fortran, Basic, LISP, Prolog, and Forth.

5) A capability that permits prototyping of 80376 embedded applications on 80386 personal computers and extensions to the I/O level with the addition of the 82370 integrated system peripheral chip. Since the 82370 provides a superset of the I/O peripherals found on an IBM PC AT or compatible, design engineers can develop a very fast prototyping vehicle. They can do this by building an AT add-in board that replicates the I/O functions of the target embedded system. This board, along with the 80386-based AT compatible, is close enough to the final target system that most of the software development work can be done directly on the PC. This development technique eliminates the downloading and debugging of code from the development system to the target system. Consequently, the final integration of hardware and software is simplified, which in turn means lower overall development cost.

6) A 32-bit programming model that provides protection and debugging features for the software engineer who needs high performance in an embedded system design. On-chip debugging capabilities greatly simplify the difficult and time-consuming software task of finding and removing flaws in new programs.

7) A segmentation model that supports segments up to 16 megabytes in length.

Table 1.
The different packages available for the Intel 376 family of components.

Part	Available packages
80376	100-pin, fine-pitch; 88-pin PGA
80387SX	68-pin PLCC
82370	100-pin, fine-pitch; 132-pin PGA
27C203A	40-pin DIP; 44-pin PLCC; 44-pin PGA; 44-pin CLC

Packaging. As can be seen in Table 1, each member of the 376 family comes in surface-mountable plastic packages and standard Pin Grid Array, or PGA, packages that aid prototyping. Because they were designed to work together, they interconnect easily. Figure 1 shows the way the components interconnect to form a 16-MHz, zero-wait-state system that includes:

- a 32-bit processor,
- 32 kilobytes of EPROM,
- 64K bytes of static RAM,
- eight DMA channels,
- 15 external interrupts,
- five 16-bit timers, and
- a serial controller.

A complete system built with these components takes up 13.5 square inches of space, an attractive size since embedded applications often need systems with very compact packaging. Figure 2 illustrates the size of a surface-mounted board required for this system example.

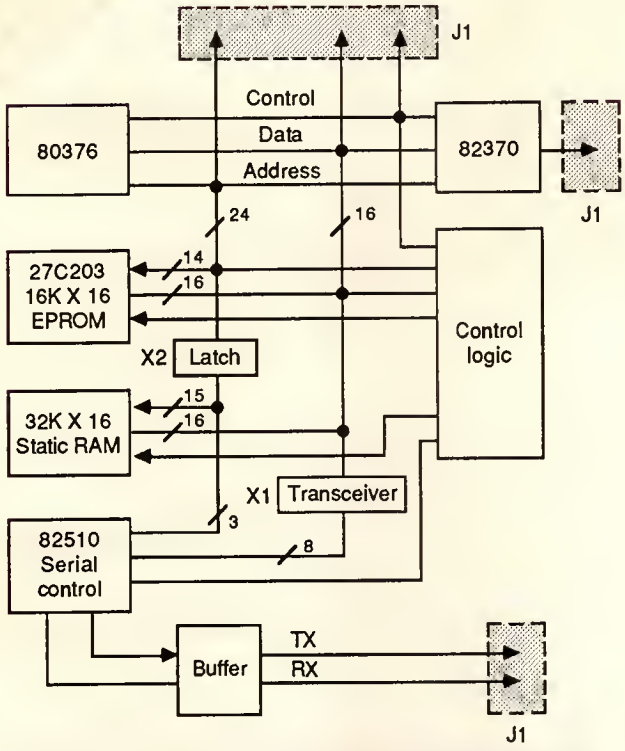


Figure 1. A typical system using the Intel 376 family of components.

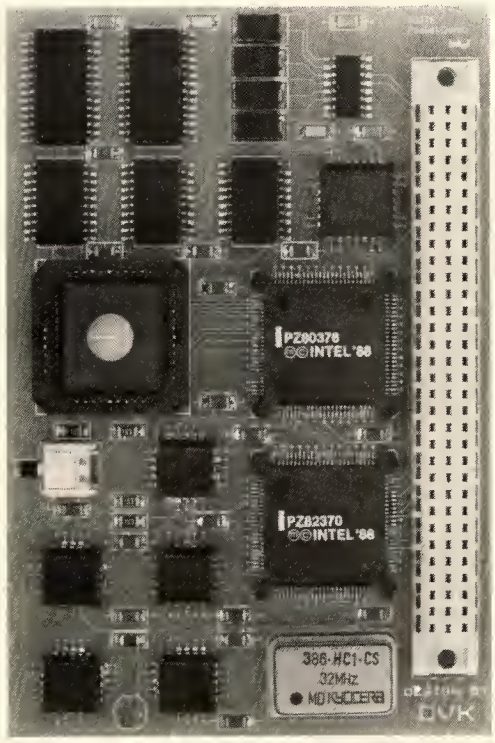


Figure 2. A surface-mounted board for the system shown in Figure 1; note the size of the board.

The 376 architecture. The 80376's 32-bit microarchitecture (seen in Figure 3) optimizes the bus interface unit for a 16-bit data path. The prefetcher fetches 16 bytes ahead, on 16-bit word boundaries. Thus the 80376 runs faster on a 16-bit bus than does the 386 microprocessor. The decode unit decodes instructions into the processor's microcode and then holds up to three decoded instructions in a queue. The execution unit, the heart of the processor, executes the instructions. The 64-bit-wide barrel shifter (important for key graphics operations) gives a very fast (average about 1.5 milliseconds at 16 MHz) integer multiply time. The user-selectable segmentation unit checks types and validates pointers. Protection checks occur in parallel with code execution without performance penalty.

Register set. As can be seen in Figure 4, the 376 processor contains

- eight general-purpose registers,
- six segment registers,
- six debugging registers, and
- various status and control registers.

The eight general-purpose registers are highly orthogonal. All the common instructions (adds, moves, multiply, Boolean operations) can use any one of these registers as the source or destination for an operation. Some specialized instructions (string moves, double shifts, divides) use dedicated registers. In this way, specialized instructions can be smaller and execute faster than they would when using a general-purpose register. (See the box on the next page for further discussion.)

A valuable feature of the processor is the orthogonality of register usage in address computation. The 80376 can use each one of the eight general-purpose registers as a base register. And, it can use any register (except ESP) as an index register for computing effective addresses.

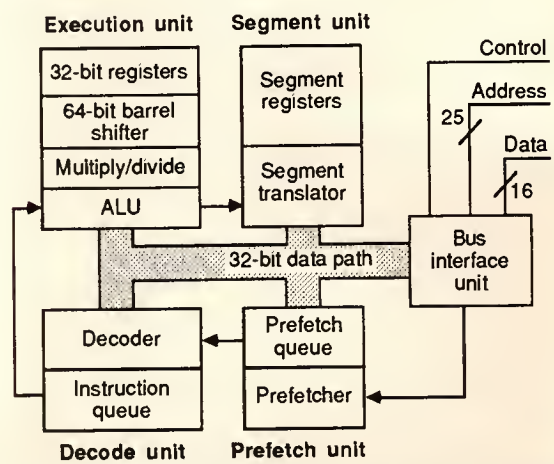


Figure 3. The 376 processor microarchitecture.

The registers allow small instruction encoding. For example, several of the most commonly used instructions (PUSH, POP, INCrement, DECrement) are only one byte long. This compact instruction encoding reduces the bus bandwidth spent prefetching instructions, which minimizes EPROM requirements and the need for instruction caches.

The 80376 passes parameters to a subroutine via a two-cycle bus with fast (a PUSH or POP of any 32-bit register takes 250 nanoseconds at 16 MHz) access to the stack. For the rare programs that require more than eight registers, the 80376 saves 32-bit registers and restores them on the stack in 10 clocks (four for the PUSH and six for the POP). This step has a negligible effect on code size since the PUSH and POP instructions each take one byte.

Addressing modes. The 80376 implements 11 addressing modes, which consist of three components:

- a base register,
- an index register with a scaling factor (1, 2, 4, or 8), and
- a displacement factor.

The addressing modes support the typical addressing requirements of high-level languages (HLL). The addressing modes strike a balance between simple load and store operations provided by RISC (reduced instruction set computer) processors and the complex addressing modes supported by many CISC (complex instruction computer) processors. Designers balanced simple load and store operations with complex addressing modes to compensate for inefficient or complex operations. Simple load and store operations inefficiently support HLL because they require additional instructions to compute the address of an operand. Most of the complex addressing modes are rarely used and result in increased compiler complexity with little performance improvement.

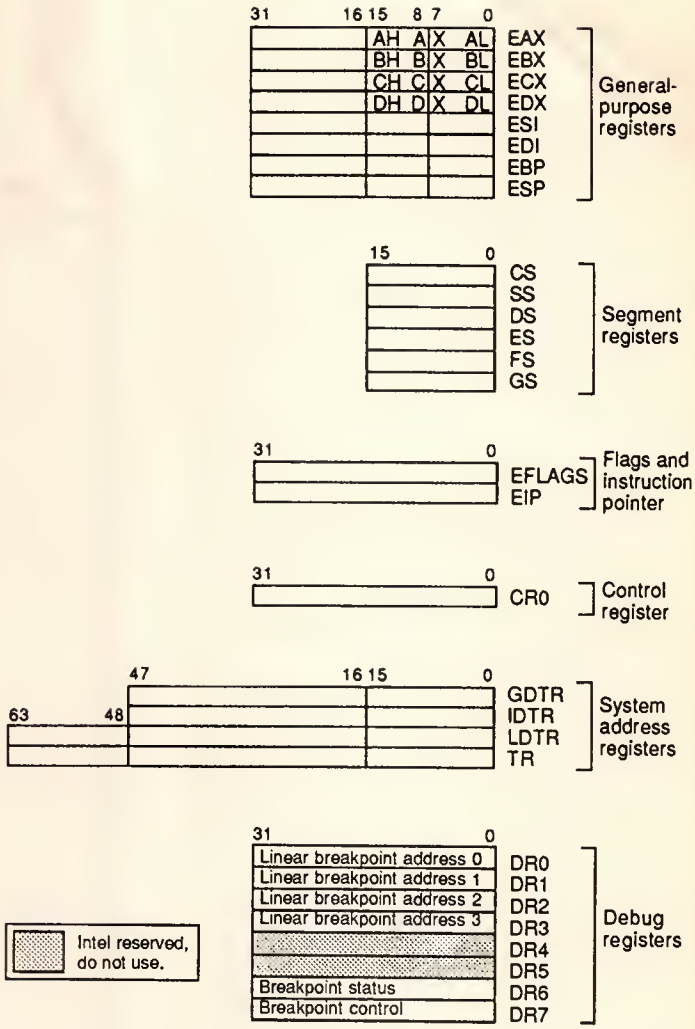


Figure 4. Base architecture registers.

Are Eight Registers Enough?

Does performance suffer if we use only eight general-purpose registers? We explored academic studies that examined the benefits of using eight registers instead of 16 when processors used stack-based parameter passing.

A 1977 study by Amund Lundee[1] determined that 90 percent of the time eight registers were sufficient for assembly language programs, and 98 percent of the time eight registers were sufficient for optimized compilers. Lundee also found that expanding to 16 registers typically achieved a performance gain of five to 15 percent, depending on the language and the program. More importantly, the study found that the most optimized compiler suffered only a one percent performance degradation when it reduced the number of registers from 16 to eight. Optimization techniques decreased the need for a larger number of registers by using register resources more efficiently.

Ten years later, another study by Flynn et al.[2] came to similar conclusions about the minimal advantages of using more registers versus increasing the instruction set complexity. This study found that allocating registers across procedures was useful, while parameter passing through registers required at least 32 registers to be effective.

The addressing modes allow a one-to-one mapping of common HLL statements to instructions. For example, the scaled index address component accesses byte, word, double, or quadword arrays in one instruction.

Execution speed is just as important as the amount of addressing modes. On the 376 processor, effective address calculation typically requires no additional time. Special hardware, such as a three-input adder and a barrel shifter, produces a worst-case effective address calculation (simultaneous use of a base, scaled index, and displacement) in one additional clock cycle. The use of complex addressing modes on some other 32-bit processors results in significant performance penalty, which may be as much as 24 clock cycles.

Instruction set. The 376 processor is compatible with the 32-bit orthogonal instruction set of the 386 processor. Designers had several reasons to choose an existing instruction set for a processor targeted at midrange embedded applications. The primary reason, discussed earlier, was the availability of development tools on many different hosts.

Another reason for choosing the 80386's native mode instruction set was its space efficiency. In reprogrammable applications the instruction set code size is of secondary importance to performance since additional memory can easily be added to the system. However, in embedded applications designers must consider memory cost. Many 32-bit embedded applications have 0.5M to 1M byte of code and 1M to 2M bytes of dynamic RAM. Therefore, an increase in code size can significantly affect the overall cost of the system.

The differences in instruction set code space needed by different processors can be significant. For example, Berkeley's RISC II requires 25 percent more space than a Digital Equipment Corporation VAX,[3] and Sun Microsystems' Sparc chip needs 50 percent more than the Motorola 68020. A 250 percent increase occurs between a RISC subset of the VAX instruction set and the full VAX.[4] Compared to most other 32-bit processors, the 80376 instruction set uses space very efficiently since it has a large number of instructions that require only one or two bytes. An internal study using an instruction mix representative of large applications found that the average 80376 instruction requires 3.2 bytes.

The instruction set falls into two categories, *core* and *specialized*. Core instructions are those that are common to most computers, ADD, MOVe, SUBtract, MULTiply, AND, OR, and so on. They have one or two operands, one of which must be located in a register and another that can be located in either a register or memory. Core instructions are compact (1 to 3 bytes) and fast (typically two clock cycles for register-to-register operations and two to eight clock cycles for memory-to/from-register operations.) The orthogonality of the core instructions permits the compiler to generate highly optimized code.

Specialized instructions include string operations, bit manipulation, 64-bit operations, loop instructions, and various conversion instructions such as those for changing ASCII to decimal. These instructions significantly enhance the performance of the inner loop of an application by replacing multiple core instructions with one instruction.

One specialized instruction is smaller and generally faster than the sequence of core instructions that it replaces. For example, one of the most useful specialized instructions is the repeated move string instruction (REP MOVSD), which copies a block of memory from one location to another. This 2-byte instruction takes eight clock cycles per 32 bit item to execute. It replaces six instructions that take 13 bytes and 24 clock cycles

Table 2.
Repeated string move equivalent.

Clocks	Bytes	Instruction	Explanation
4	1	PUSH EDX	; Save contents of EDX
		Loop1:	
6	2	MOV EDX, [ESI]	; Load the next four bytes into EAX
4	2	MOV [EDI], EDX	; Store it into location pointed
			; to by EDI
2	3	ADD ESI, 4	; Increment starting address
2	3	ADD EDI, 4	; Increment destination address
2	1	DEC EXC	; reduce copy count by one
8	2	JNZ loop1	; loop back until the transfer
			; is complete (ECX = = 0)
		POP EDX	; restore contents of EDX

per 32-bit item. (See Table 2.) This string move operation keeps the bus busy 100 percent of the time. At 16 MHz with zero-wait-state pipelined memories, data moves at 6.4M to 8M bytes per second, depending on operand alignment.

Embedded applications particularly benefit from the use of specialized instructions that optimize time-critical sections. Most embedded applications contain several routines in which the performance is so important that the routines are written in assembly language. Specialized instructions work well in assembly language modules. One example is the bitblt operation. The accompanying box illustrates how the performance and code size of this common graphics routine can be enhanced by two specialized instructions: double shift and string move.

The Bitblt

Bitblt (bit block transfer) is a fundamental graphics operation that copies a fixed-size bit image from one region to another. Neither image needs to be aligned on a byte, word, or dword boundary. Typical embedded applications using bitblts include laser printers, graphics terminals, and optical scanners.

Figure A shows the operation of a bitblt and the code needed to accomplish it. Table A lists the clocks and bytes involved. Note that:

- The 64-bit barrel-shifter lets each 32-bit data item be aligned properly in one three-clock instruction.
- Each one-byte, common instruction for a load and store string (LODS, STOS) replaces two instructions (a move and an add), thereby saving one clock cycle and 3 bytes on each use.
- The entire operation involves more computing than busing, as might be expected for a bitblt operation. Because the instructions are so compact (1.5 bytes per instruction), sufficient time exists to prefetch all the instructions before they are needed—even on a 16-bit bus. This core routine results in a peak bitblt rate of 17M bits/s.

Processors with a 32-bit barrel shifter typically must deal with 16-bit data items, even if they have a 32-bit bus.

Table A.
Assembly language program for a bitblt operation.

	Clock cycles	Length (bytes)
MOV ESI, Source address	7	1
MOV EDI, Destination address	3	3
MOV EBX, Word count	3	1
MOV CL, Relative offset	6	1
MOV EDX, [ESI]	2	1
ADD ESI, 4	9	2
Total	**30**	**9**

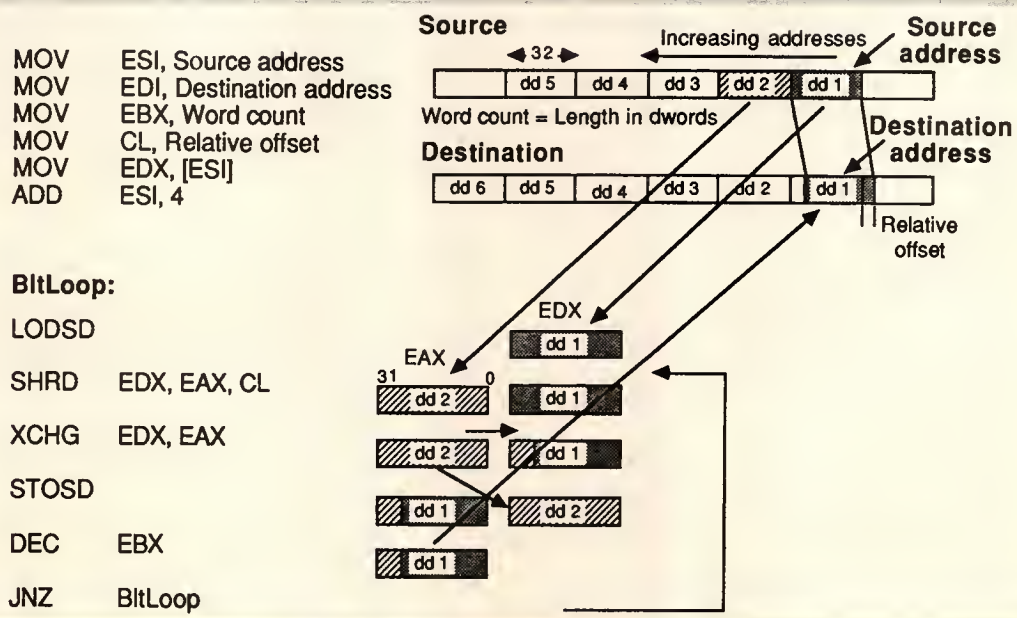


Figure A. 376 processor code for a bitblt kernel.

Protection. The built-in protection and segmentation mechanisms in the 376 processor allow physical memory to be managed efficiently in multitasking systems. These segmentation-based protection mechanisms let the system have complete control over memory use, an important factor in some sophisticated applications.

The majority of embedded applications require little or no memory management and are well supported by an unprotected linear address space. Accomplished via a short (five instructions) initialization sequence, the application sets up one code and a data segment that can be as large as the physical memory.

However, some sophisticated embedded applications require a highly reliable, protected, multitasking environment. For these applications the 376 processor offers a number of robust, hardware-enforced protection mechanisms:

1) four privilege levels;
2) call gates to control privilege-level transfers;
3) privilege verification of pointers;
4) hardware support for multitasking;
5) intratask protection through separate code, data, and stack regions; and
6) complete intertask isolation and protection.

Software development tools support these protection mechanisms.

Even the simplest embedded application can benefit from protection features during the product development cycle as several problems can occur while debugging embedded systems. For example, a bad pointer can overwrite code causing mysterious crashes that are difficult to find. Also, stack overflows can be difficult to detect, and continual checking degrades performance. Finally, "buggy" applications tend to corrupt the RAM-based debugger.

The 376 processor protects memory locations involved in these problems against corruption by making each one a separate segment. With this type of protection, we simplify debugging because bad pointers and stack overflows immediately cause a general protection violation. Debuggers locate an error quickly since the address of the faulting instruction and an error code are pushed onto the stack before control passes to the exception handler. At a minimum, debuggers can be installed in a protected region of memory to guarantee that control passes to the user under all error conditions. Most of the available debuggers use processor protection mechanisms.

Debug support. In addition to the protection features, designers added several debugging aides for quicker development of embedded applications:

- Debuggers can use a one-byte debug instruction to generate instruction breakpoints. The debugger replaces the first byte of an instruction with an INT3 opcode. When the breakpoint is reached, and the 376 processor generates an Interrupt 3 condition with vector 3, the processor passes control to the Interrupt 3 handlers (typically, a debugger).
- Debuggers can initiate an instruction single-stepping procedure by setting the TF bit in the flag registers. When this bit is set, the 376 processor generates an Exception 1 condition after the execution of each instruction. Control then passes to the Interrupt 1 handler (typically, a debugger).
- Debug registers provide the most flexible aid. As mentioned earlier, the 376 processor contains six of these registers. Four of these registers (DR0 through DR3) specify four distinct, 32-bit linear addresses that cause the breakpoint. DR6, the breakpoint status register, provides information to the debugger about the breakpoint reached. DR7, the control register, specifies the breakpoint conditions.

In contrast to traditional breakpoint instructions that only permit breakpoints to be set for RAM-based instruction execution, the 80376 debug registers offer breakpoints under conditions such as:

- instruction execution (not just prefetch),
- data writes,
- data reads or writes,
- local to a task, and
- global to all tasks.

This flexibility is important for embedded applications because breakpoints can be set in ROM-based systems, thus enhancing the utility of software debuggers for real-time systems. For instance, Microsoft Corporation designed its CodeView debugger so variables could be monitored via a Watch command. At first, CodeView executed slowly when using the command. Tracking variables is extremely useful, but observing a variable with special code checks greatly slows down the execution speed of a program—often an unacceptable condition for real-time applications. Later, Microsoft enhanced CodeView to make use of debug registers. Now, users "watch" variables at close to real time.

Hardware for the 376 processor

When coupled with the 82370 high-integration system peripheral chip, the 376 processor becomes the core of a midrange embedded system that is cost effective to develop and manufacture. The two-chip set requires one quarter the board space needed by most other 32-bit microprocessor systems. Major features of the hardware include:

- sustained execution of 2.5 million to 3.0 million instructions per second at 16 MHz;
- a 32-bit programming model with a 16M-byte physical memory;
- a 32-bit instruction set compatible with the 386 microprocessor instruction set;

• four on-chip breakpoint registers;
• optional, segmentation-based protection and memory management;
• a demultiplexed, 16-bit data bus and a 24-bit address bus with a 16M-byte/s bus bandwidth;
• address pipelining for zero-wait-state operation with 100-nanosecond DRAMs;
• direct connection to the 82370 chip without additional logic, and
• packaging in either a 100-lead, surface-mountable, gull-wing flat pack or an 88-lead, ceramic pin grid array.

Figure 5 shows the buses and the control and status signals making up the 376 processor hardware interface, and Table 3 (on the next page) describes the functions of pertinent pins.

Clock and control signals. The double-frequency, 32-MHz clock CLK2 drives the 80376. This clock is internally divided by two and synchronized to form the internal processor clock signal PCLK. We refer to the low and high periods of PCLK as phases 1 and 2 and each PCLK period as a T-state. All bus cycles start at the beginning of phase 1 and stop at the end of phase 2. Figure 6 (on page 19) shows the relation of CLK2, PCLK, and some bus control signals in a typical zero-wait-state bus cycle.

The 80376 can generate eight different types of bus cycles. To indicate to the external hardware the kind of bus cycle that is occurring, the 80376 asserts status signals M/IO#, D/C#, W/R# and the address bus at the start of a bus cycle. Table 4 decodes these signals into the different bus cycles.

Nonpipelined bus cycles. The processor bus consists of a 16-bit, bidirectional data bus; a 24-bit address bus consisting of 23 address pins and 2-byte select signals; plus status and control signals. The basic, nonpipelined bus cycle operates in different states, as shown in the state diagram in Figure 7 (on page 19).

Ti is the idle state, the state of the bus logic after a RESET signal and the state to which bus control logic returns after every nonpipelined cycle.

T1 is the first state of a nonpipelined bus cycle. The 387 drives its address bus and status signals at the beginning of every T1 state. If the bus is in a write cycle, the 387 drives the data bus during phase 2 of T1.

T2 is the second state of a nonpipelined bus cycle. At the end of phase 2 of this cycle, the 387 samples the READY# signal to determine whether to terminate the bus cycle or to add an additional T2 state. If the bus cycle is terminated and it is a read cycle, the 387 samples the data bus and deactivates the address and status signals (or drives the next address and status signals if a bus cycle is pending) at the end of phase 2. If it is a write cycle, the 387 deactivates the address and status bus (or drives the next address and status signals if a bus cycle is pending) at the end of phase 2. For write cycles, the 387 deactivates the data bus during the following bus cycles, T1, phase 2.

Th is the hold state, the state entered in response to a HOLD request and exited when the HOLD signal is deactivated. When in this state the 387 tristates all of its output signals except the HLDA signal, which is activated to a high level. This condition allows other bus masters in the system to gain control of the bus.

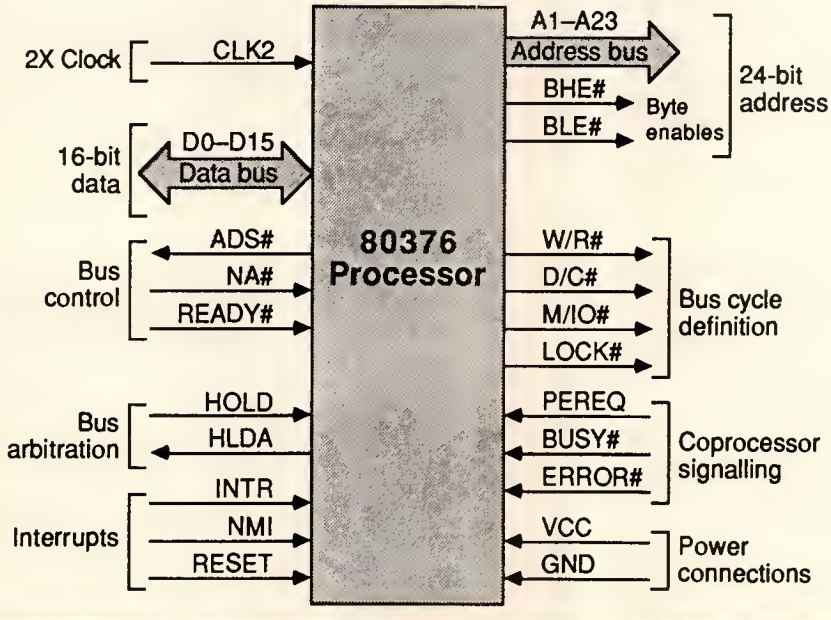


Figure 5. Typical 80376 bus cycle.

Table 3.
80376 pin description.

Symbol	Type	Name and function
CLK2	I	CLK2 provides the fundamental timing for the processor.
RESET	I	RESET suspends any operation in progress and places the processor in a known reset state.
D15-D0	I/O	DATA BUS inputs data during memory, I/O and interrupt acknowledge read cycles and outputs data during memory and I/O write cycles.
A23-A1	O	Address Bus outputs physical memory or port I/O addresses.
W/R#	O	Write/Read is a bus cycle definition pin that distinguishes write cycles from read cycles.
D/C#	O	Data/Control is a bus cycle definition pin that distinguishes data cycles, either memory or I/O, from control cycles which are: interrupt acknowledge, halt, and, instruction fetching.
M/IO#	O	Memory/IO is a bus cycle definition pin that distinguishes memory cycles from input/output cycles.
LOCK#	O	Bus Lock is a bus cycle definition pin that indicates that other system bus masters are denied access to the system bus while it is active.
ADS#	O	Address Status indicates that a valid bus cycle definition and address (W/R#, D/C#, M/IO#, BHE#, BLE#, and A23-A1) are being driven at the pins.
NA#	I	Next Address requests address pipelining.
READY#	I	Bus Ready terminates the bus cycle.
BHE#, BLE#	O	High and Low Byte Enables indicate which data byte of the data bus takes part in a bus cycle.
HOLD	I	Bus Hold Request input allows another bus master to request control of the local bus.
HLDA	O	Bus Hold Acknowledge output indicates that the processor has surrendered control of its local bus to another bus master.
INTR	I	Interrupt Request is a maskable input that signals the processor to suspend execution of the current program and execute an interrupt acknowledge function.
	I	NMI (Non-Maskable Interrupt Request) is a nonmaskable input that signals the processor to suspend execution of the current program and execute an interrupt acknowledge function.
BUSY#	I	Busy signals a busy condition from a processor extension.
ERROR#	I	Error signals an error condition from a processor extension.
PEREQ	I	Processor Extension Request indicates that the processor has data to be transferred by the 376 processor.

Table 4.
Bus cycle definition.

M/IO#	D/C#	W/R#	Bus cycle type
0	0	0	INTERRUPT ACKNOWLEDGE
0	0	1	Does not occur
0	1	0	I/O DATA READ
0	1	1	I/O DATA WRITE
1	0	0	MEMORY CODE READ
1	0	1	HALT: Address = 2, BHE# = 1, BLE# = 0 / SHUTDOWN: Address = 0, BHE# = 1, BLE# = 0
1	1	0	MEMORY DATA READ
1	1	1	MEMORY DATA WRITE

Figure 8 presents examples of nonpipelined bus cycles. Note that the T2 state repeats for requested wait states by not asserting the READY# signal. The address and control signals remain active for the entire bus cycle. For small systems the address bus can be connected directly to the address pins of peripheral devices.

Pipelined bus cycles. To take advantage of slower main-memory chips, the bus uses a control input called Next Address (NA#). External hardware uses this input to switch from the bus's two-clock access to a slightly slower pipelined mode. This mode increases the address access time without decreasing the data bus bandwidth. Address pipelining occurs when the address of

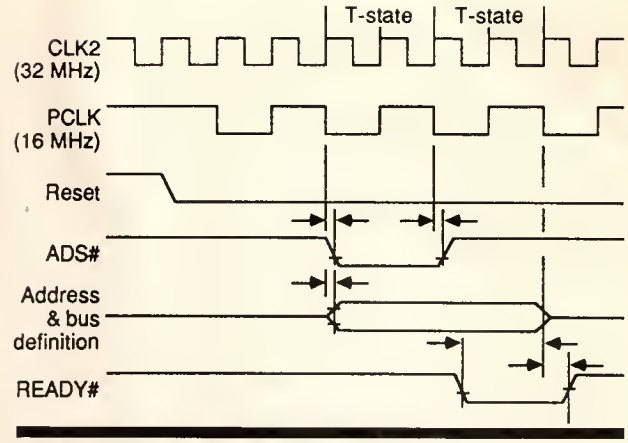


Figure 6. Bus states (not using pipelined address).

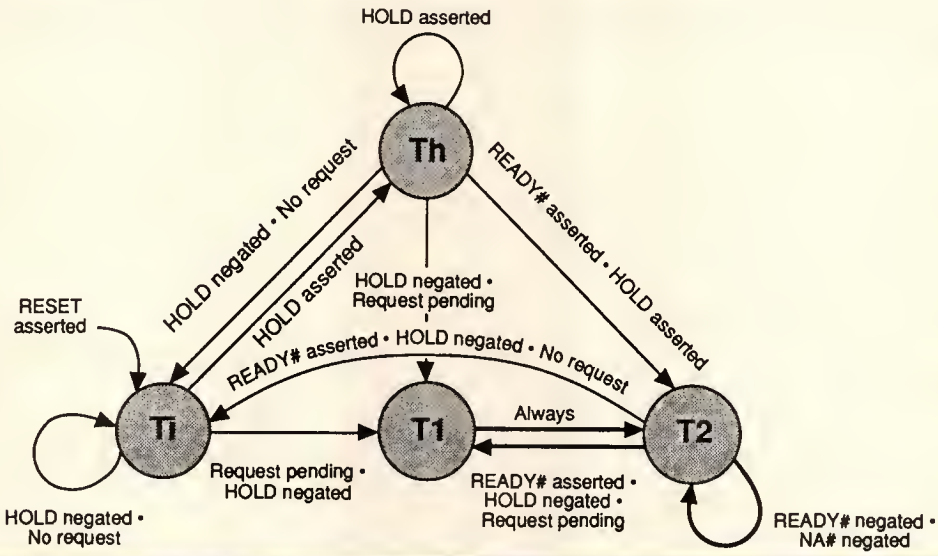


Figure 7. 80376 nonpipelined bus cycles.

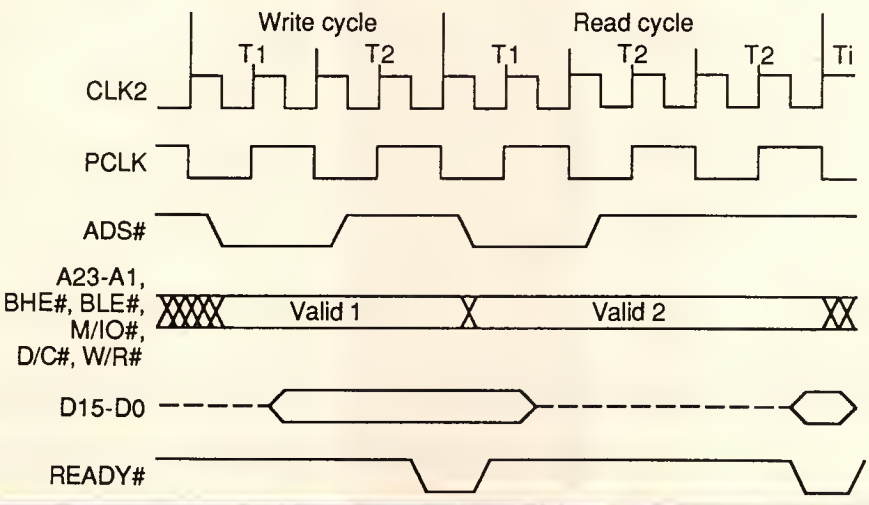


Figure 8. Complete 80376 bus states.

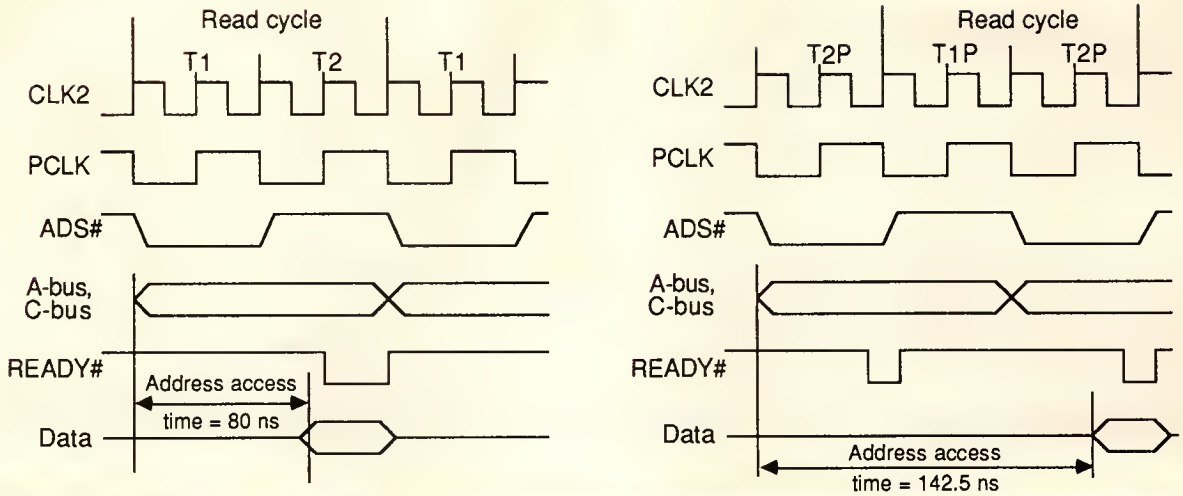


Figure 9. Pipelined bus cycles.

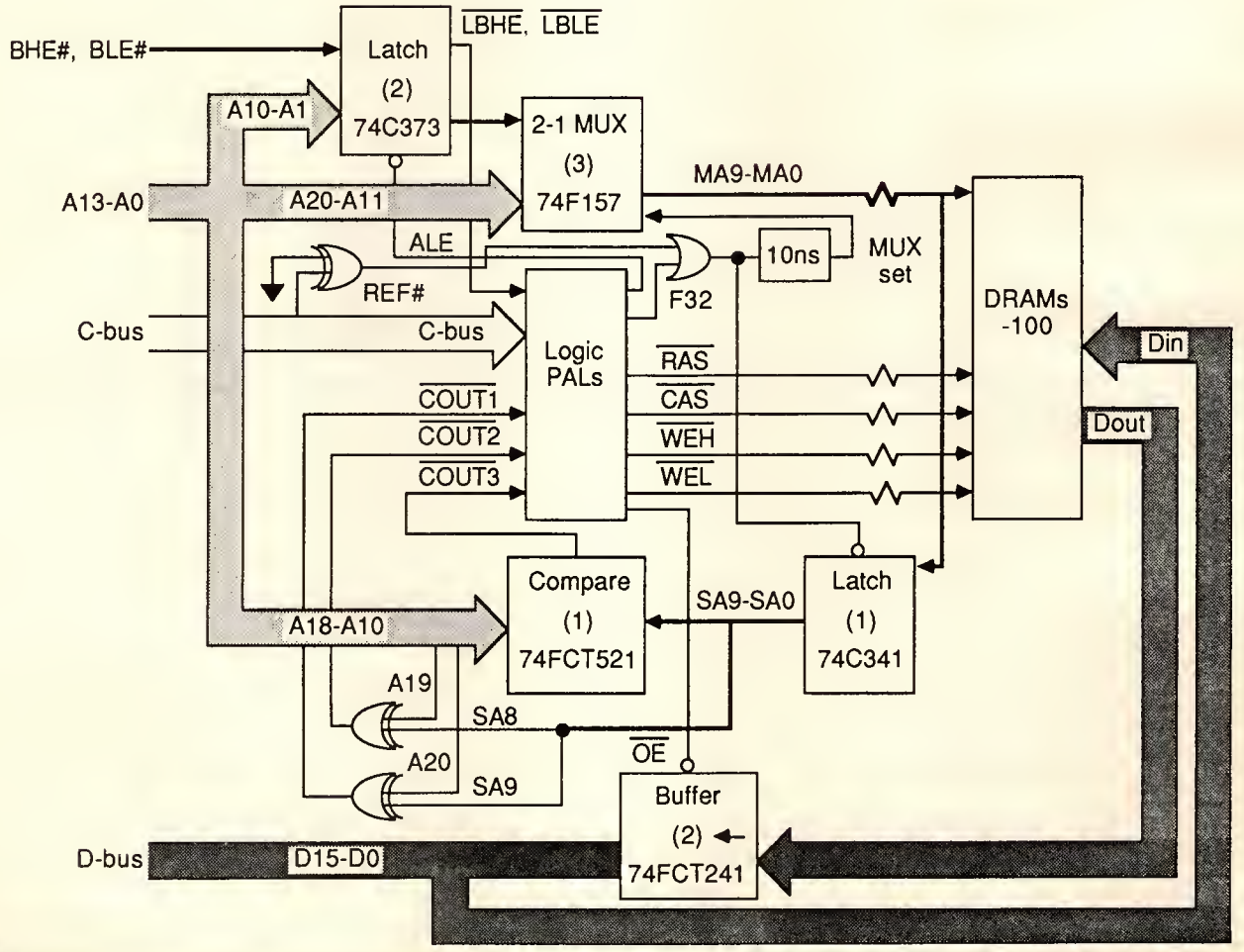


Figure 10. Example of a page-mode DRAM interface.

the next bus cycle overlaps the data already on the bus during the last clock cycle of the current bus cycle. Thus the pipelined cycle has a latency of three clock cycles from address to data but uses only two clock cycles on each of the subsequent address and data transfers to maintain the 16M-byte/s peak bandwidth. The addition of pipelining supplies three additional states to the bus state machine.

Because pipelining requires one additional T-state for the 376 processor to enter from an idle cycle, the performance of a system using a pipelined address bus is slightly less than that of a nonpipelined bus. This performance degradation, however, is greatly reduced because the 376 processor maintains the pipelined bus cycle. The cycle is maintained because the 376 processor uses approximately 90 percent of the available bus bandwidth and tends not to go into the idle state.

The high bus use results from a number of things:

- *Number of bytes per instruction.* The average opcode for an instruction is 3.2 bytes long; that is, bringing a typical instruction into the 80376 requires 1.6 code fetches. Because prefetch requests have a low priority in the bus unit, these prefetch cycles only execute when the data bus would normally be idle, or when the prefetch queue is empty and the execution unit is waiting for an instruction. (Examples would be after a reset state or after a transfer of control such as a JUMP or CALL instruction.)
- *Size of the prefetch queue.* The 16-byte prefetch queue provides a balance between filling up what could have been idle bus cycles with code fetches and letting data transfers occur as quickly as possible.
- *Size of data accesses.* Because many transfers are 32-bit quantities, the 80376 requires two memory accesses.

The combination of these three factors results in consistent, back-to-back bus cycles, enabling the pipelined bus to execute at its maximum bandwidth before being disabled by an idle cycle. Over 90 percent of its memory cycles occur in the pipelined mode if the NA# signal allows it.

The system follows a simple process to request pipelining. Address latches hold the previous bus cycle's address and status while another device receives the address and status of the next bus cycle. In Figure 9 we see that pipelining increases the address time needed for the memory to respond by one clock cycle (62.5 ns). The increase from 80 ns to 142.5 ns is especially important when considering the much lower cost of the slower memory devices.

Interfacing DRAMs. With the hardware interface we can build a variety of different-speed DRAM interfaces. Of course, the major objective for most systems is balancing the cost of the DRAM design with its performance. In this case, the best design has the best cost-to-performance ratio. For example, a zero-wait-state, nonpipelined memory design gives the highest performance possible but requires either static memories with access times of 80 ns or DRAMs with access times of 40 ns. A pipelined, zero-wait-state design gives the next-highest performance but requires memories with 100-ns access times and 125-ns cycle times. A page-mode DRAM design provides the next-highest performance, using DRAMs from 120 ns to 80 ns. However, the memory design that results in the best cost-to-performance ratio is a page-mode DRAM design using a 100-ns DRAM.

Figure 10 shows the components needed to implement a high-performance, page-mode DRAM design using 100-ns DRAMs. Table 5 lists the wait-state penalties that occur when this particular DRAM interface is used. This memory design complements the

Table 5.
Page-mode wait-state penalties.

DRAMs Bus usage estimate (percent)	Description	Wait states
From an idle bus cycle		
5	Read	1
2	Write	1
From a pipe cycle		
30	Read hit	0
2	Read hits after read miss	1
2	Read hits after other	2
10	Write hit	0
3	Write hits after read miss	0
5	Write hits after other	1
From other device pipe cycle		
25	Read	0
12	Write	0
Back-to-back nonpipe cycles to same device		
1	Read after read miss	1
1	Read after other	2
1	Write after read miss	0
1	Write after other	1
Average wait states		**0.22**

pipelined EPROM design in two ways. Because accesses to the EPROM are pipelined, back-to-back memory accesses that switch from DRAM to EPROM occur with no wait-state penalty. In addition, back-to-back memory accesses that switch from EPROM to DRAM also occur with no wait-state penalty.

The 82370 integrated system peripheral

The 82370, a multifunction system support peripheral, integrates the necessary processor support functions required in an embedded environment. It consists of several computer system functions that are normally found in separate LSI (large-scale integration) and VLSI (very large scale integration) components. The interface requires little or no TTL (transistor-transistor logic) "glue" since the 82370 was specifically designed to work with the 80376. Also, the 370 has the same AC timings and address and data buses. Figure 11 provides a block diagram of the chip, which features:

- optimized use with the 376 embedded processor;
- a high-performance (32-bit internal bus) DMA controller;
- a 16-bit data path, 24-bit addressing, and eight independently programmable channels;
- a 16M-byte/s data transfer rate at 16 MHz;
- a 20-source interrupt controller, the 82C59A superset;
- 15 external and five internal interrupts plus individually programmable interrupt vectors;
- four 16-bit, programmable, 82C54-compatible interval timers;
- a programmable wait-state generator with zero to 15 wait states;
- a DRAM refresh controller;
- 80376 shutdown detection and reset controls with software and hardware resetting; and
- CHMOS III technology.

DMA controller. The 82370's DMA controller supports eight DMA channels, each of which is capable of peak transfer rates of 16M bytes/s in a 16-MHz system. Each channel transfers data between devices of different data path widths and operates independently in any of several modes.

The 82370 achieves a high transfer rate by using the fly-by mode, one of the many special DMA modes on the 82370. In this mode the 82370 reads and writes data in the same two-clock bus cycle. (Normal DMA transfers use two bus cycles requiring four clocks: two for a read cycle and two for a write cycle.) Two bus-cycle transfers permit simple hardware design since the access to the I/O device and memory work just like a CPU access. Table 6 lists the different operating modes available for the 82370's DMA controller.

Each of the eight DMA channels contains separate DMA request lines (DREQ*n*) through which external devices request service. Request-line input can be programmed to respond to either a synchronous or an asynchronous signal. When servicing a DMA call, the 82370 identifies the active DMA channel (excluding channel 4) on a 3-bit DMA ACKnowledge (EDACK)

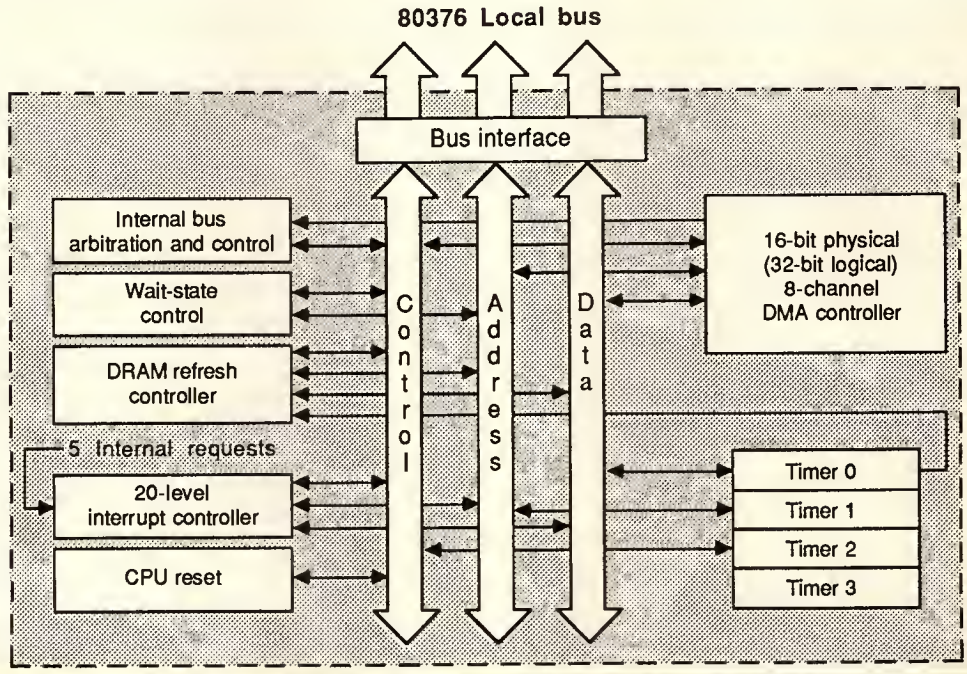


Figure 11. Block diagram of the 82370 integrated system peripheral chip.

Table 6.
DMA operating modes on the 82370 controller.

Target/requester definition
- Data transfer direction
- Device type
- Increment/decrement/hold

Buffer processes
- Single-buffer process
- Buffer auto-initialize process
- Buffer-chaining process

Data transfer/handshake modes
- Single-transfer mode
- Demand-transfer mode
- Block-transfer mode
- Cascade mode

Priority arbitration
- Fixed
- Rotating
- Programmable fixed

Bus operation
- Fly-by (single cycle)/two cycle
- Data path width
- Read, write, or verify cycles

bus during accesses to the requester. (Note that channel 4 does not have a corresponding hardware acknowledge.) Either external hardware directly monitors this bus to determine the device being serviced or an external decoder generates separate DMA ACKnowledge signals for each of the eight channels. Table 7 lists the DMA acknowledge codes.

Additionally, the DMA controller contains an end-of-process signal. This EOP# signal acts in one of two different ways. As an output, it indicates to external hardware when the last bus cycle of a DMA process occurs. As an input, it also indicates to the 370 that the external hardware does not require additional data transfers for this channel. This capability is very useful, for example, when transferring frames of data to an Ethernet controller. If the Ethernet controller detects a "collision," it can assert the EOP# signal. The 82370 will then reinitialize its registers and resend the frame, without CPU intervention.

The 82370 contains four 16-bit, programmable interval timers. These timers are identical to the timers in the 82C54 programmable interval timer. All four timers share a common clock input (CLKIN) driven by an external oscillator. Each has a dedicated output pin. Timer 1's dual-purpose output can be programmed to generate normal 82C54 timer outputs or to generate a

Table 7.
EDACK encoding during a DMA transfer.

EDACK2	EDACK1	EDACK0	Active channel
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	Idle or target access
1	0	1	5
1	1	0	6
1	1	1	7

refresh signal for DRAM subsystems. Outputs for timers 2, 3, and 4 internally connect to interrupt request lines so these timers can generate timing interrupts to the processor. The timers are capable of operating in six different modes. In all of the modes, the current count can be latched and read by the processor at any time.

Programmable interrupt controllers. The 82370 has the equivalent of three enhanced 82C59A programmable interrupt controllers. Users can access 15 external interrupt request inputs, all of which can be inputs from external slave interrupt controllers. By cascading 82C59A controllers to these inputs, a possible total of 120 external interrupt requests can be allowed. Interrupt request lines internal to the 82370 let timers, DMA chaining requests, or terminal counts generate interrupts.

Wait-state generator. The wait-state generator is a programmable (READY#) generation circuit for the processor bus. With software, designers can program seven different wait states into the generator: three for memory accesses, three for I/O accesses, and one for refresh cycles. One register contains the desired number of wait states for each of the seven wait-state counts. An external device selects the number of wait states by placing the address of its predefined register on the wait-state bus (WS0-WS1). This 2-bit address is decoded along with the M/IO# signal and the REF# signal to select the appropriate register. An address of 3 disables the wait-state generator.

DRAM refresh controller. The 82370 DRAM refresh controller consists of a 24-bit refresh address counter and bus arbitration logic. Timer 1's output periodically requests refresh cycles. When timers activate the controller, it requests access to the system bus through the HOLD signal. When the processor's bus master grants bus control, the refresh controller executes a memory read cycle at the address currently in

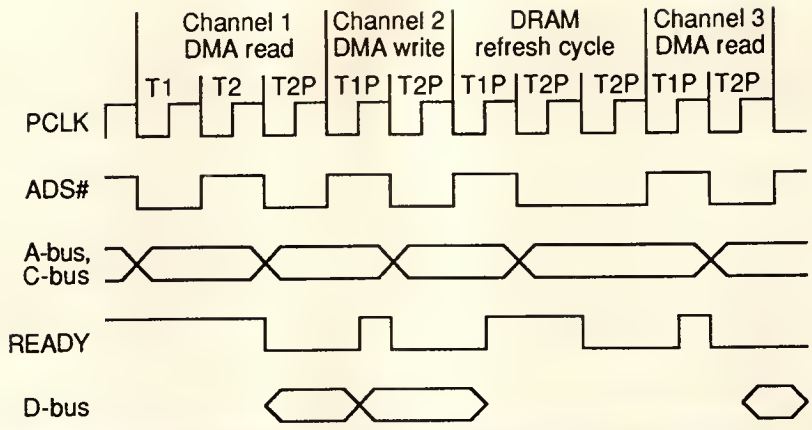


Figure 12. Example of 370 bus utilization.

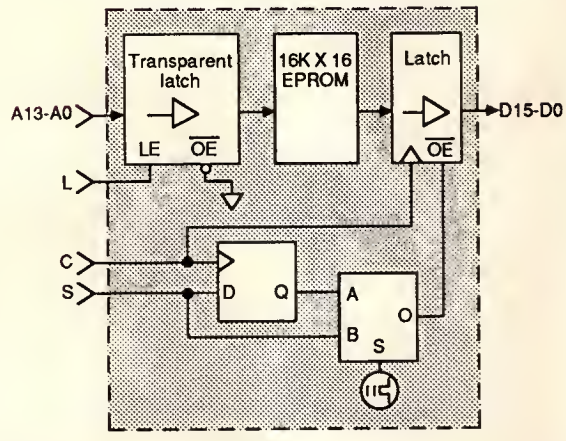


Figure 13. Internal block diagram of the 27C203A pipelined 256K-byte EPROM.

the refresh address register. Figure 12 shows how the 82370 gains control of the bus, executes back-to-back DMA cycles, executes a refresh sequence, and then returns the bus to the processor. Efficient use of the system bus maximizes performance of the host CPU.

The 27C203A EPROM

The 27C203A optimizes 80376 embedded applications with an internal pipeline that speeds the execution rate for sequentially accessed program code. The pipeline achieves higher bandwidths than are possible with a standard interface. Figure 13 is an internal block diagram of this part. The EPROM contains a pipelined interface of a clock for data latching and synchronous chip selection and supports the 16-MHz, pipelined 376 processor. Other features include:

- CHMOS technology of 256K density with a maximum current of 110 milliamperes,
- a drive capability of a 4-mA source/16-mA sink current,
- a programming algorithm with 4-second throughput for automated manufacturing,
- several package options, and
- a standard 27C202A interface option.

The 27C203A eliminates the need for external address latches that are normally associated with pipelined address buses (refer to the box on the next page). It contains an on-chip transparent latch for the address (not used for 376 processor applications), as well as one latch for its data bus. The use of the data latch allows data to remain valid at the outputs while the next bus cycle address is placed on the address pins.

The 27C203A interface consists of the address pins; data pins; and three control signals, address latch input L, clock input C, and select input S. With the 376 processor, only the C and S control signals are used. The clock signal latches the data on the output pins. After the appropriate address setup time has been met (55 ns for a 27C203-55), a rising edge of the C input latches the data on the output pins. If the select signal is low during the rising edge of C, the data appears on the output pins. If the output signal is high during the rising edge of C, the output data buffers go into a high-impedance state.

The Pipelined Address Bus

The 80376 pipelined bus adds access times for bus cycles. In the case of synchronous memory designs (such as DRAM controllers), this additional access time lets all the decoding occur before the actual bus cycle begins. Also, the state machines can trigger at the beginning of the bus cycle (T1P).

Asynchronous devices (such as EPROMs) generate valid data output as a result of a valid address input. Because the pipelined bus changes the address before the end of the cycle, the address normally must be latched. A latched address provides a valid address during the entire bus cycle. This step has the disadvantage of letting the address start propagating to the EPROM only at the beginning of the cycle. The 27C203A EPROM gets around this problem by latching the data outputs once the address setup time is met. This early access allows the address to change and to start meeting the address setup time before the current bus cycle is over.

Other nonlatched EPROMs can use this same approach to gain maximum access time from the pipelined bus cycle. By adding a latch to the data output buffers and a flip-flop to control the output enable of this latch, we can connect normal 100-ns EPROMs to the 80376 bus and achieve zero-wait-state performance. By adding additional wait states, we can use slower EPROMs. Table B summarizes this technique.

Another approach to gain maximum access time from the pipelined bus is to interleave banks of EPROMs. This type of system requires at least two different EPROM banks, each with its own address latches. As one EPROM bank is being accessed, the pipelined address propagates to the other EPROM bank. The low-order address bit A1 controls the selection of the EPROM bank. Consecutive prefetch cycles access alternate banks.

Interleaving EPROMs allows zero-wait-state pipelined accesses using 120-ns EPROMs. Additional wait states permit the use of slower EPROMs. Table C illustrates the EPROM requirements of an interleaved system. The timing comparison showing the extra clock cycle of address time was shown in Figure 9.

Table B.
EPROM requirements for a pipelined bus using a latched data output.

Wait states	Standard EPROM speed (ns)
0	100
1	150
2	200
3	250
4	350

Table C.
EPROM requirements for a pipelined bus using an interleaving scheme.

Wait states	Standard EPROM speed (ns)
0	120
1	200
2	250
3	300
4	350

The 80387SX numeric coprocessor

The 80387SX numeric coprocessor provides additional numeric processing power to those embedded applications needing fast floating-point support. Systems benefiting from this numeric coprocessor include robots and numerical control applications. The coprocessor features:

- an 80-bit internal architecture;
- IEEE 754 floating-point standard conformance;
- data types such as 32-bit single real, 64-bit double real, 80-bit extended real, 16-bit word integer, 32-bit short integer, 64-bit long integer, and 18-digit BCD integer;
- object-code compatibility with the 8087 family;
- optimized interface with the 376 processor for highest possible floating-point performance;
- direct extension of the 376 processor instruction set to include trigonometric, logarithmic, exponential, and arithmetic instructions for all data types; and
- overall performance of 1.3 million double-precision Whetstones per second.

The Intel 376 family helps the embedded system designer develop applications that require large amounts of processing at low cost in a small space. The high integration and small physical size of the 32-bit 376 processor coupled with the 82370 integrated system peripheral chip make it possible to build a complete embedded system with RAM, EPROM, numerics, DMA, timers, and interrupt controllers on a 4 × 5-inch board. Together, the 80376 and the 82370 deliver performance of 2.5 to 3.0 MIPS without the expensive, high-speed caches needed in a low-cost, 16-bit system.

Additionally, designers can eliminate up to 30 LSI and VLSI components that are needed in most other 32-bit systems. They can use standard 100-ns DRAMs in building 80376 systems that run close to zero-wait-state performance. And they can take advantage of the processor's easy connection to many inexpensive 8-and 16-bit components.

As a subset of the 80386 architecture, the 376 processor provides several benefits that can result in lower development costs. First, the compact encoding of its instruction set reduces code size and consequently memory requirements. Second, the 80376 uses the many development tools and real-time kernels already in existence for the 386 processor. Finally, the high degree of compatibility between the two processors means that standard workstations can be used as hardware and software prototyping vehicles for 376 processor embedded applications.

Clif Purkiser is Intel Corporation's product manager for the 376 embedded processor. He has worked with 386 microprocessor marketing and application activities for three years. He received his BS in electrical engineering and computer sciences from the University of California at Berkeley and an MBA from Santa Clara University.

Jim Kardach is an applications engineer in the microprocessor group. He received a BS in electrical engineering from California State University, Fresno and is a member of Eta Kappa Nu and Tau Beta Pi.

Questions concerning this article can be addressed to Jim Kardach at Intel Corporation, Microprocessor Group, MS SC4-40, 2625 Walsh Avenue, Santa Clara, CA 95052-8122.

References

1. A. Lundee, "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," *Comm. ACM,* Mar. 1977.
2. M.J. Flynn, C.L. Mitchell, and J.M. Mulder, "And Now a Case for More Complex Instruction Sets," *Computer,* Sept. 1987, pp. 71-83.
3. D. Patterson, "Reduced Instruction Set Computers," *Comm. ACM,* Jan. 1985.
4. J. Davidson and R. Vaughan, "The Effect of Instruction Set Complexity on Program Size and Memory Performance," *Comm. ACM,* Jan. 1987.

Additional reading

El-Ayat, K., and R.K. Agarwal, "The Intel 80386 Architecture and Implementation, *IEEE Micro,* Dec. 1985, pp. 4-22.

80386 Programmer's Reference Manual, Intel Corporation, Santa Clara, Calif., 1986.

Hennessy, J., "VLSI Processor Architecture," *IEEE Trans. Computers,* Dec. 1984.

1989 International Editorial Calendar

IEEE Micro helps designers and users of microprocessor and microcomputer systems explore the latest technologies to achieve business and research objectives.

Feature articles in **IEEE Micro** are original works relating to the design, performance, or application of microprocessors and microcomputers. Tutorial material, industry views, and standards developments are also published.

AD CLOSING DATE: 1st of month preceding issue (Jan. 1st for February issue). Contact Advertising Director Dawn Peck, Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578; (714) 821-8380.*

HOW TO SUBMIT AN ARTICLE TO IEEE MICRO

Prospective contributors should request Author Guidelines from:

James J. Farrell III
Editor-in-Chief,
IEEE Micro
VLSI Technology, Inc.
8375 South River Parkway
Tempe, AZ 85284
(602) 752-6222

All manuscripts are subject to a peer-review process consistent with prefessional-level technical publications. **IEEE Micro** is a bimonthly publication of the Computer Society.

*Articles may change. Please contact editor to confirm.

FEBRUARY
Applications
The Racer's Edge: Advanced applications using microprocessors
Submit by: 7-15-88

APRIL
Special Far East Issue
The latest technology in Japan, Korea, Taiwan, and the Pacific Basin—microprocessors, support devices, new systems, standards, and innovative applications
Submit by: 9-15-88

JUNE
General Interest
Submit by: 11-15-88

AUGUST
Memory Technology, CAD, and CIM
Submit by: 1-15-89

OCTOBER
Special European/ Near East Issue
Annual display of the most exciting microelectronic technology on and off the continents
Submit by: 3-15-89

DECEMBER
General Interest
Come to us with a micro-related topic
Submit by: 5-15-89

FEATURE

Processor Architecture Considerations for Embedded Controller Applications

The VL86C010 ARM processor improves real-time I/O and interrupt-latency operations to cut hardware requirements in embedded controller applications.

Ron Cates
VLSI Technology, Inc.

Computer hardware architects face a never-ending task: to provide increasingly higher performance and utility to users at ever-decreasing cost. For example, today's desktop systems furnish virtually the same performance that mainframe computers did just a few years ago. However, as small-system performance has increased, these systems have also encountered the same bottlenecks as their larger counterparts. And, in general, similar means have proven instrumental in solving both sets of problems.

Today's microprocessors encompass a great deal of the architecture developed for mainframe computers five years ago. These microprocessing units (MPUs) can operate with shorter cycle times than dynamic RAM (DRAM) devices provide. To solve this bottleneck, most MPU devices contain some type of cache memory implementation to increase memory bandwidth and allow the processor to function more closely to its full-speed operation. Similarly, as performance has increased, applications programs have grown correspondingly more complex, with larger code size and more data. As a result, many MPUs have integrated full virtual-memory support to provide larger address space and allow programs that are independent of the size of real-system memory.

Small systems have recently encountered yet another bottleneck: input/output-bound tasks. Small systems usually have a single processor element that is responsible for controlling all functions within the system. This processor performs I/O operations when required. Traditionally, performance increases have resulted strictly from enhancements made to the processor element. This approach worked because the processor was still the limiting factor in system performance. Now many systems spend more time waiting for data to be transferred from the disk than they do processing the information. In these cases, supplying a faster disk drive would do more to increase system performance than using a faster processor. In fact, a faster processor makes no difference at all in some cases.

On the other hand, a faster disk drive may be prohibitively expensive. Many times it is more cost-effective to use extra hardware that makes a slow disk drive *appear* to be faster, much the way cache memory makes DRAM *look* faster. Unfortunately, cache memory controller ideas do not directly extend to the disk section.

To be effective, the disk controller requires much more intelligence than the cache controller. While one may implement a cache controller with a fairly simple state-machine, the disk controller requires an intelligent com-

072-1732/88/0600-0028$01.00 © 1988 IEEE

puting element dedicated to the task of increasing disk-drive performance. When a processor is dedicated to a control task and the user is unaware of the processor's existence (the user does not program it directly), this processor becomes an embedded controller.

Many designers are currently adding embedded controllers to their system architectures to increase performance. New special-purpose processors are debuting with architectures that better match the controller environment than they do the role of general-purpose processor. This trend is particularly evident in the area of graphic display controllers.

Environmental differences

The environment of an embedded controller usually differs from that of a general-purpose CPU. In reality, the environment contains many additional embedded processors because most processors designed especially for CPU applications cannot adequately cope with the real-world interface. For example, many modern processors based on the complex instruction set computer (CISC) methodology rely on microcode implementations requiring a number of clock cycles to complete. Designers and users of these processors feel that the rich, complex instruction set offers better performance than processors that have simpler instruction sets. An unfortunate by-product of this methodology is that the processor cannot be interrupted during these long microcode sequences, which makes the system unresponsive to external events. To deal with asynchronous events, many systems use an embedded processor that can better respond to the environment as a preprocessor for the main CPU. One can then optimize the intelligent subsystem towards desired characteristics.

A processor designed for embedded controller applications generally possesses a narrower focus of performance considerations than other processors. One can remove unnecessary hardware and replace it with special-purpose logic more useful to the assigned task. Even though some logic may increase performance, it might not be employed with enough frequency to justify its presence. The differences between microprocessor and microcomputer architectures demonstrate this fact. Most MPU families support floating-point calculations, whereas MCUs do not address this support in hardware. By removing support for infrequently used functions, the designer can add hardware such as barrel shifters that assist in bit manipulation and data reformatting—tasks that embedded controllers usually perform. In addition, one can slightly change other areas of the architecture to increase the functionality for controller applications that the CPU generally finds difficult to use. Examples are extra registers for dedicated tasks or a simple, single-level looping construct.

Here, we examine an embedded controller application: the network interface. We first assay the application environment to determine the processor characteristics required to match the environmental parameters. Such system trade-offs occurred when Acorn Computers of England designed the VL86C010 Acorn RISC (reduced instruction set computer) Machine (ARM). The design result was a full 32-bit processor that performs equally as well in both embedded controllers and general-purpose processor applications. Acorn developed the VL86C010 on CAD software from VLSI Technology, Inc. with six man-years of total effort.[1] This processor and its three associated peripheral circuits function as the main elements of the Archimedes desktop computer that Acorn manufactures in the United Kingdom.

Network controller applications

Most small systems today require networking to share and exchange information in a simple and efficient manner. To accommodate a fairly large number of nodes—up to 1000—on the network, most protocols operate at 1-10 megabits per second. However, the nodes do not require this high data rate for large amounts of continuous data exchange. In fact, most traffic is short and burst-like in nature. In addition, internal buses in most desktop computers cannot sustain these data rates for very long without noticeable performance degradation. High data rates minimize the packet-delivery time of any two communicating nodes while all the other nodes in the system wait for them to finish. The network must respond to the needs of many users so that multiple nodes can share the cost of the media installation.

The system designer usually adds a network-embedded controller for one reason: to minimize the visible performance degradation of the node. Two factors accomplish this goal: additional processor resources and a better match of processor bus requirements to network demands. This match occurs through the use of a dedicated interface buffer to store the data and then forward it when the network is available (store and forward buffering).

Here, we present a network interface architecture with an environment that has three attributes:

- asynchronous packet arrival,
- very high bandwidth demands during packet delivery, and
- simple computational requirements from finite-state, machine-protocol definitions.

A network of small computers exhibits interactive behavior. Network messages arrive and are dispatched at variable intervals. Nodes may have no network traffic for long periods of time and then suddenly become quite active for a while during a long file transfer. This pattern of activity is the reason that most networks use packet switching as opposed to the circuit switching used in voice applications.

During a "voice call," (a) the bandwidth require-

ments are fixed and continuous, and (b) call-processing overhead is short compared to the length of the call. In contrast, interactive network traffic would not require a fixed bandwidth and would produce a significant call-processing overhead. Packet switching provides a better match of traffic-to-bandwidth requirements than circuit switching.

Once the packet begins, the bus must provide a fairly high bandwidth for the transfer. In the case of a 10M-bit network and a 16-bit-wide processor bus, a data word must be provided every 1.6 microseconds on the average. If the memory cycle is 400 nanoseconds, then 25 percent of the bandwidth is dedicated to the network during the transfer, assuming a DMA device does the transfer. If the processor uses programmed I/O in place of the DMA channel, then 100 percent of the bus bandwidth can be employed during network operations. Note that this transfer may be totally unrelated to the program operating on the node or even anything initiated on the node by the user, that is, data sharing across the network or a system with multiple file servers. The remote processor consumes some of the local performance, which slows the current task proportionately.

Modern networks ordinarily provide reliable transport services to protect against the loss or duplication of data. Using the protocol definition on the media accomplishes this protection. Most protocols append a header on each packet to transfer some state information to the receiving node. The receiving node examines this information and determines if the state is correct relative to the defined protocol. The protocol defines all actions taken regarding the state information. As a consequence, network controllers must efficiently perform bit manipulation, data movement, and conditional branching.

Network architectures

As discussed, the designer determines which characteristics of the processor will enhance its performance in this particular embedded controller application. In a network environment, two main considerations are processor interrupt latency and processor ability to provide efficient support for high-bandwidth transfers. Both of these characteristics require large amounts of extra hardware if the chosen processor does not provide them.

Interrupt latency considerations. Three components determine interrupt latency:

- maximal delay of the instruction set,
- fixed software interruption overhead, and
- fixed hardware interruption overhead.

For most CISCs, the instruction-set component is the largest component to consider, software overhead comes in second, and hardware overhead is least significant. The instruction latency is the time required to synchronize the event with the instruction boundary clock. This clock usually maintains a fairly low frequency in a CISC. Interrupting an instruction once it begins execution is exceptionally difficult because the various pipeline stages must be saved and restored to maintain partial results. For example, consider a CISC multiply instruction. Say, it is implemented using a Booth's Two-Bit Algorithm that multiplies two 16-bit numbers with a 32-bit result in a maximum of eight clock cycles. This process requires a 32-bit temporary holding register inside the ALU that can hold the partial product during calculation. In cases of instruction interruption, the processor saves this temporary register in addition to the information regarding the amount of the instruction completed. When long execution pipelines are present, the amount of status that must be saved grows rapidly: If the instruction set delay is reduced, the software overhead grows. Most CISCs find that it is simpler and usually just as fast to allow interrupts only on instruction boundaries. In the case of the Motorola MC68020, the longest instruction is DIVS.L (with the most complex addressing mode) at 119 clock cycles.[2] Even with a 25-MHz clock, this component alone takes 4.76 μs.

Software overhead is customarily the second largest component of interrupt latency. This software overhead value relates to the amount of processor state information that must be preserved before interrupt processing can begin. Most processor specifications require only the minimum amount of state preservation rather than a full context switch of the processor. This minimum amount equals the time required to stack the following values:

- program counter,
- processor status register, and
- any other local state information that a task can control.

This software overhead component is a usual function of the memory access time that allows for stacking information in memory to provide recursiveness in interrupts and code sections. Designers should seek to minimize this time to avoid wasting execution resources, that is, reducing the time available for useful computation. In embedded controllers, very high interrupt frequency can become a significant problem.

In addition, one must consider hardware overhead, which characteristically takes the form of request synchronization. This synchronization function occurs much more quickly than the instruction set latency because it usually relates to the highest clock frequency available. Hardware overhead can be as simple as a single clock edge or as complex as a resynchronization of several related internal-state machines. Requests must be retimed to the internal clock to ensure reliable operation of the processor. This process also minimizes the impact of metastable failure (violation of setup and hold times on synchronous elements) in the system. In general, this hardware overhead time is not as critical

as the other two components because it is overlaid onto the instruction pipeline delay. Therefore the time specified is the same amount required for beginning pipeline resynchronization.

For the MC68020, the combined hardware and software overhead amounts to 33 clock cycles,[2] assuming a no-wait-state memory. The processor must stack two 32-bit words of internal state, each word requiring a three-clock memory cycle. This requirement brings the total worst-case interrupt latency to 152 clock cycles at 40 ns each, or a total of 6.08 μs.

Embedded processors must deal with asynchronous events efficiently and in a cost-effective manner because each main CPU may have several of these intelligent subsystems. If a processor has a long interrupt latency, then one must add extra hardware to support real-time requirements while the processor is recognizing the state change. Of course, extra hardware adds to the cost of the system. In addition, the latency reduces computing bandwidth available for useful work. Suppose a processor receives an interrupt event every 100 μs for short periods of time and it must complete its task before the next cycle. The MC68020 would have only 93.92 μs to perform the task because it must complete within the required worst-case time, which equals interrupt frequency (100 μs) minus worst-case latency (6.08 μs). Because the system must perform in all cases under worst-case analysis or risk unreliable operation, embedded controllers generally cannot fully utilize special structures such as cache memory or FIFOs.

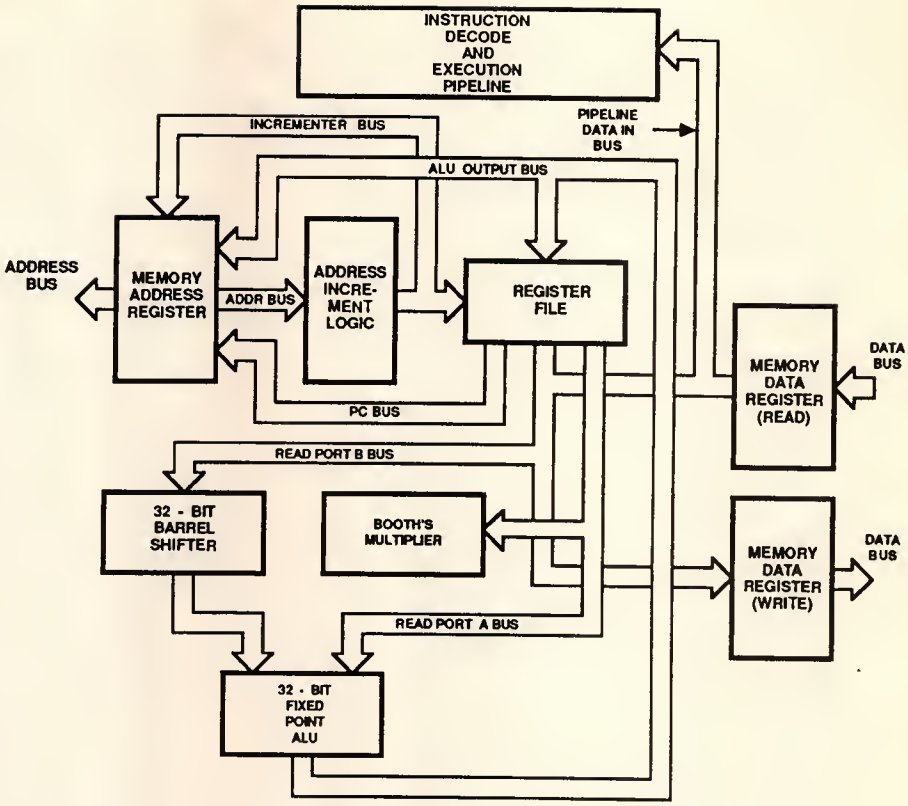


Figure 1. VL86C010 block diagram.

Architectural issues. We designed the VL86C010 ARM processor to maintain a real-time interrupt latency. This feature allows the processor to handle I/O operations directly and minimizes the need for extra hardware.[1] We optimized several areas of the processor to retain its responsive bus:

- instruction-set implementation,
- register-file organization,
- interrupt-vector handling, and
- memory-bus operation.

The result is a processor with a worst-case interrupt latency of 22.5 clock cycles, or 1.87 μs at 12-MHz operation. Figure 1 is a block diagram of the processor and Figure 2 is a functional pin diagram. The processor die size is a very compact 230 mils × 230 mils in 2 μm, two-layer, metal process rules.

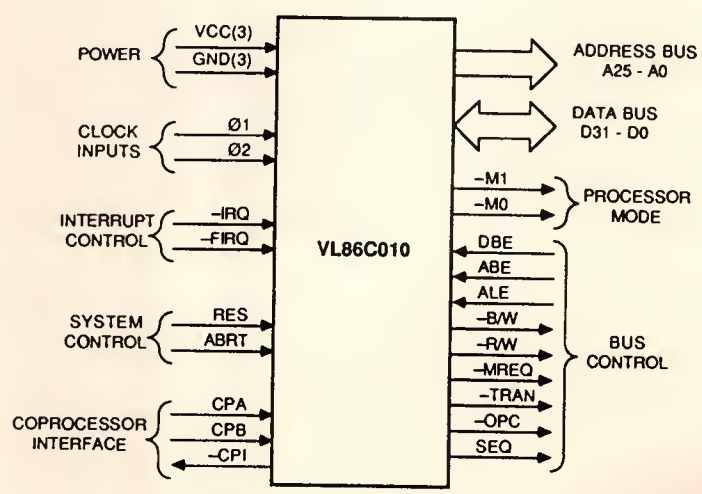


Figure 2. VL86C010 functional pin diagram.

Table 1. VL86C010 instructions.

Function	Mnemonic	Operation	Processor cycles†
Data processing			
Add with carry	ADC	Rd: = Rn + Shift(Rm)*+ C	1S
Add	ADD	Rd: = Rn + Shift(Rm)	1S
And	AND	Rd: = Rn • Shift(Rm)	1S
Bit clear	BIC	Rd: = Rn • Not Shift(Rm)	1S
Compare negative	CMN	Shift(Rm) + Rn	1S
Compare	CMP	Rn - Shift(Rm)	1S
Exclusive—OR	EOR	Rd: = Rn XOR Shift(Rm)	1S
Multiply with accumulate	MLA	Rn: = Rm × Rs + Rd	16S max
Move	MOV	Rn: = Shift(Rm)	1S
Multiply	MUL	Rn: = Rm × Rs	16S max
Move negative	MVN	Rd: = NOT Shift(Rm)	1S
Inclusive—OR	ORR	Rd: = Rn OR Shift(Rm)	1S
Reverse subtract	RSB	Rd: = Shift(Rm) - Rn	1S
Reverse subtract with carry	RSC	Rd: = Shift(Rm) - Rn - 1 + C	1S
Subtract with carry	SBC	Rd: = Rn - Shift(Rm) - 1 + C	1S
Subtract	SUB	Rd: = Rn - Shift(Rm)	1S
Test for equality	TEQ	Rn XOR Shift(Rm)	1S
Test masked	TST	Rn • Shift(Rm)	1S
Data transfer			
Load register	LDR	Rd: = Effective address	2S + 1N
Store register	STR	Effective address: = Rd	2N
Multiple data transfer			
Load multiple	LDM	Rlist: = Effective address	(n**+1)S + 1N
Store multiple	STM	Effective address: = Rlist	(n+1)S + 2N
Jump			
Branch	B	PC: = PC+Offset	2S + 1N
Branch and link	BL	R14: = PC, PC: = PC+Offset	2S + 1N
Software interrupt	SWI	R14: = PC, PC: = Vector #	2S + 1N

* Shift() denotes the output of the 32-bit barrel shifter. It shifts an operand in several ways on every data processing instruction without requiring additional cycles.

** n is the number of registers in the transfer list.

† S denotes sequential memory cycle.
N denotes nonsequential memory cycle.

Instruction-set latency. Table 1 lists the instructions supported by the VL86C010 as well as the associated number of processor cycles required for each instruction. All data processing instructions execute in one clock cycle except MULTIPLY, which requires a maximum of 16 clock cycles. Processor cycles consist of two types: nonsequential N and sequential S. In a system using DRAM devices, an N cycle requires two system clock cycles while an S cycle requires one. A system that uses static memory can have both cycles of equal length—usually one clock cycle each—as the processor does not require very fast memory in either case. The processor supports the basic operations in hardware only to simplify and minimize the number of gates, shorten processor cycle time for increased performance, and move functionality into software for improved flexibility. We used a load/store architecture that supports data processing only on the processor registers and provides separate instructions for memory reference operations. In general, this type of overall architectural definition seems to support efficient implementation of compilers for high-level lan-

Table 2. Small-computer benchmark results.

Program	Acorn Archimedes A310* 8 MHz	PC's Limited 386[16]** 16 MHz	IBM PS/2 Model 80** 16 MHz	Compaq Deskpro 386* 16 MHz	Apple Mac II † 15.67 MHz
Dhrystone	4901	4378	3626	3748	2083
Fibonacci	52.4	46.78	57.26	53.1	83.7
Sieve	5.7	5.33	6.45	6.0	16.7
Sort	10	6.63	7.74	5.6	22.4
Savage	91.2	21.69	9.49	21.5	5.4

* D. Pountain, "The Archimedes A310," *Byte*, Oct. 1987, pp. 125-130.
** M.L. Van Name, "The PC's Limited 386[16]," *Byte*, Dec. 1987, pp. 141-144.
† G.M. Vose, "Head to Head," *Byte*, Aug. 1987, pp. 113-114.

guages (HLLs) better than more complex computers do.[3] As shown in Table 1, the longest instruction is a data transfer to memory (LDM/STM) of all 16 visible registers and requires 18 clock cycles to complete. This value serves as the longest instruction-delay baseline for interrupt-latency calculations because the VL86C010 does not interrupt instructions once they begin execution. Compared to the MC68020, the VL86C010 instruction-set latency has been reduced by a factor of over six.

One might assume a sacrifice of performance with a reduced instruction set concept designed specifically for embedded controller applications. In reality, the VL86C010 performs benchmarks very close to the rates obtained by today's CISCs (Table 2). Note that the Archimedes obtains these performance levels employing only an 8-MHz internal clock and a 120-ns DRAM memory. The figures compare systems for several popular benchmark functions that use HLLs. In general, these benchmarks indicate both processor performance and compiler efficiency. Therefore, they provide the user with some idea of expected system performance. Please note that compiler differences can have a large impact on these performance measures.

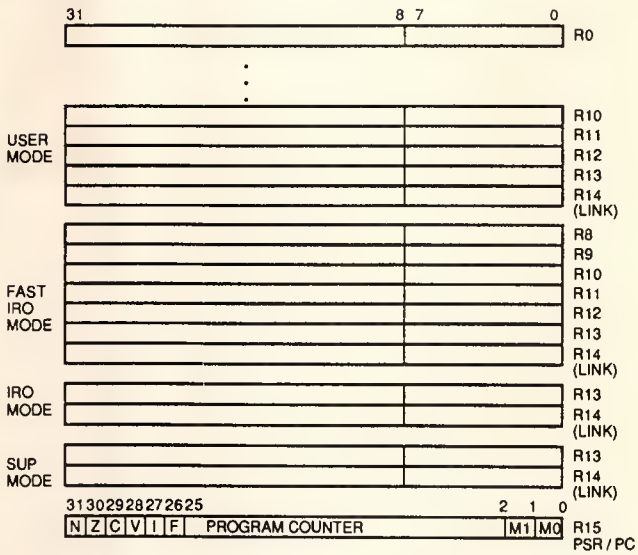


Figure 3. VL86C010 register file implementation.

Architectural overhead. As shown for the MC68020, software overhead processing can significantly affect circuit requirements. Consequently, we altered several areas of the VL86C010 architecture to minimize this parameter. The most significant impact of the alterations is in the register file definition (Figure 3). The VL86C010 contains 27 partially overlapping registers with only 16 of them visible to the program at any one time. Whenever a mode change occurs, new registers are mapped into the visible space to save context switching overhead. Many RISCs incorporate the idea of overlapping registers in their architectures to enhance processor operation. Most research work on RISC architecture design involved using a register file that overlapped the registers across procedure calls. This idea minimizes the overhead of call/return sequences used to pass parameters in modular, stack-oriented software environments. While overlapped windows do appear to assist performance of call/return sequences, they do little to aid interrupt latency, even when a new window is used for interrupt processing. The lack of dedicated registers reduces usefulness because the register file may be full when the interrupt occurs. This condition causes a register-file overflow that must be saved before the interrupt can be serviced.[4]

Table 3. VL86C010 exception vectors.

Address (Hex)	Function	Priority level
000 0000	Reset	0
000 0004	Undefined instruction trap	6
000 0008	Software interrupt	7
000 000C	Abort (prefetch)	5
000 0010	Abort (data)	2
000 0014	Address exception	1
000 0018	Normal interrupt (IRQ)	4
000 001C	Fast interrupt (FIRQ)	3

In addition, register files using this organization have to be fairly large because of unpredictable needs for procedure-nesting depth. Of course, large registers are expensive. On some processors, this requirement takes hundreds of memory cycles. If the application cannot take advantage of all the registers, then some hardware remains unused.

Overlapping VL86C010 registers across procedure calls therefore became too expensive relative to desired performance. Instead, we overlapped the registers across the processor modes. The processor supports four modes: user (16 registers), supervisor (two registers), fast interrupt (seven registers), and normal interrupt (two registers). When the mode changes, new registers are mapped into the programmer's model. One can easily dedicate these registers to support the DMA function. The interrupt handler does not need to save some of the internal registers and load the interrupt values for processing. In most cases, the entire context for a Fast Interrupt Request (FIRQ) handler can be held with the FIRQ mode registers R14__FIRQ to R8__FIRQ. This function keeps interrupt-processing overhead to a bare minimum.

By overlapping registers across modes, the VL86C010 can use R14 as a link register for both subroutine and interrupt context changes. This advantage is possible because each mode has its own private link register. Otherwise, a mode change would destroy some context in the previous mode because the processor does not directly support a hardware stack. One can implement software stacks very simply by using the addressing modes of the processor. A stack memory reference in software requires only four system clock cycles to complete—approximately the same efficiency as a hardware stack reference.

Note that one 32-bit-value R15 contains the full processor context (program counter and conditional code register). Using a link register removes any need for context-stacking sequences unless the software determines that these sequences are necessary to provide more flexibility. This link-register function can become particularly important to a very short interrupt handler that is not required to be recursive or enable interrupts during processing. If these requirements exist, then R14 can be placed on a software stack before the interrupt handler software enables the interrupts.

Exception-vector handling. We also considered interrupt latency in the layout of the vectored interrupt support. Table 3 demonstrates the vector table format. Much as in other processors, an exception condition forces a branch to a fixed location beginning with the memory location zero. The VL86C010 supports seven different exception conditions that are prioritized (highest to lowest):

- reset,
- address exception and data abort,
- fast interrupt request,
- normal interrupt request,
- prefetch abort,
- undefined instruction, and
- software interrupt.

Each vector occupies four bytes and normally contains a branch instruction to the proper location. The FIRQ vector is the last table entry to allow the handler to reside at location 000 001CH (byte address value) and above. This construction avoids the need for an associated branch instruction-execution time in this handler, which saves at least three processor cycles.

Bus issues. As discussed, memory cycle time can affect latency calculations, particularly if saving processor context requires long stacking sequences. Acorn performed a number of studies of existing processors to determine the parameters that affect performance. The results defined one dominant parameter in processor performance: the amount of available memory bandwidth.[1] This parameter holds true even when comparing older 8-bit to newer 16/32-bit processors, despite the fact that most 8-bit processors may have an inferior instruction set. Therefore, we designed the VL86C010 to exploit the maximum bandwidth available from the memory system.

Most DRAM devices support page-mode (sequential) access modes that are faster than the specified random access cycles. For example, a 120-ns DRAM device that performs a random-access cycle in 250 ns usually can provide data within the physical row of the storage array with 120-ns cycles. Unfortunately, most MPUs cannot use this mode because they themselves do not provide enough status to the memory controller. The memory controller must assume each access is random, when in reality a large portion of sequential accesses are due to program locality principles.

The VL86C010 performs benchmarks very close to the rates obtained by today's CISCs.

The VL86C010 supports page-mode transfers directly from the DRAM and so obtains approximately 50 percent more bandwidth from the same memory than other processors do. The memory bus is pipelined outside the processor so that the memory controller receives the memory control signals for the next cycle during the current cycle. In addition, the memory address bus is advanced by one half of a processor cycle to provide the maximum time for the processor to access memory. This address advance is possible because the DRAM contains address latches to hold the address stable during the access. We also provided this latch within the processor to use with ROM devices that lack their own latches. As a consequence, the memory controller has almost the entire clock cycle for access time. A 12-MHz (83-ns cycle) processor requires 80-ns memory for full-speed operation.

The sequential (SEQ) signal informs the memory controller that the next address will be the current address plus four; that is, the address incrementer will be the address source on the next cycle. Figure 4 shows the timing of the memory system bus. When the processor requests a nonsequential cycle, the memory controller performs a full access that requires two system clock cycles. When the SEQ signal is active, the memory controller can perform a -CAS-only cycle that requires only one clock cycle. Depending on how much locality the program has, this -CAS-only cycle can provide a significant performance increase.

In addition, the processor uses conditional execution of all instructions to help maximize the sequential code and prevent pipeline breaks that waste precious memory bandwidth. Besides being able to execute a BRANCH NOT EQUAL, the program can execute an ADD NOT EQUAL or LOAD REGISTER EQUAL. Instructions that are not executed require a single clock cycle to be skipped. This conditional execution helps minimize the number of branch instructions and, therefore, the number of pipeline breaks. A VL86C010 branch instruction requires only three clock cycles; consequently, a program can skip one or two instructions and operate faster than if branch instructions are used. In the MC68020, a branch instruction can take up to nine clock cycles,[2] and jumps or branches will appear somewhat more often than in equivalent VL86C010 programs.

Page-mode cycles do lower interrupt latency because the longest instruction is a multiple-data transfer of all processor registers. Page-mode cycles are used for memory-operand references as well as instruction fetches. This procedure reduces the time required for 16 memory references from 34 to 18 clock cycles and reduces the interrupt latency by the same amount.

As a result of all these considerations, the interrupt latency of the FIRQ interrupt is:

- two and one-half cycles for request synchronization,
- 18 cycles for the longest instruction, and
- two cycles for software context-switching overhead.

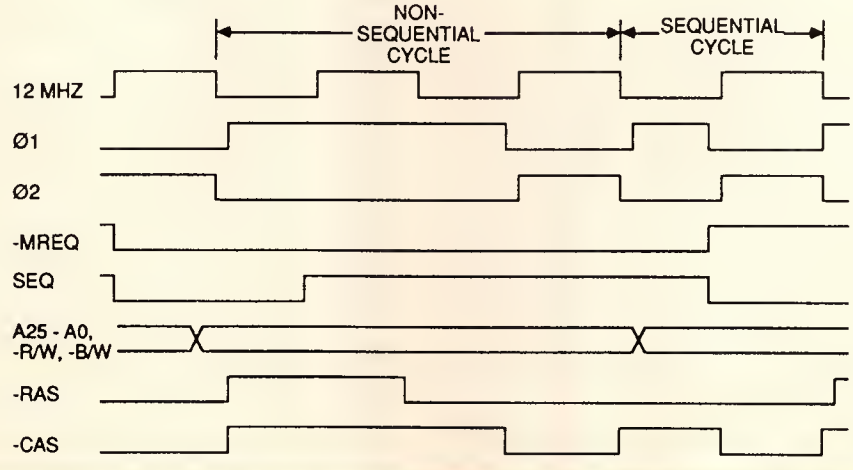


Figure 4. VL86C010 bus signal timing.

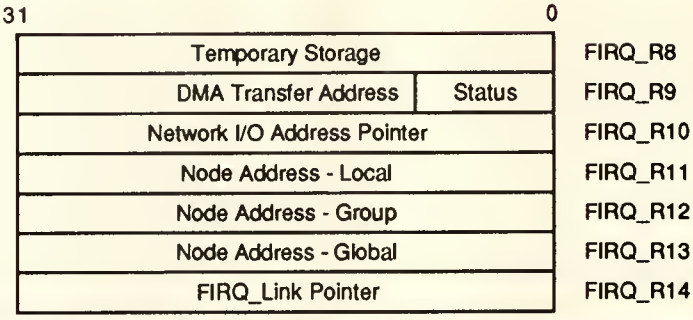


Figure 5. Fast-interrupt request register usage.

The VL86C010 obtains approximately 50 percent more bandwidth from the same memory.

The total latency time is 22.5 clock cycles, or 1.87 μs worst case, for a system using a free-running, 12-MHz clock. This time is only about one third as long as that of the MC68020. The minimum latency is one and one-half cycles for synchronization plus two cycles for the overhead for a total of three and one-half cycles, or 292 ns. A system that performs neither full stacking sequences nor multiply instructions can approach the minimum latency easily.

System implications

Whenever designers face processing data in real-time, system-latency considerations can have a large impact on which circuitry they choose. As previously stated, many times the wrong choice of processor requires additional hardware to deal with the data while the processor performs its context switch. Two of the most frequently used devices are FIFO memory and direct memory access controllers (DMACs). The correct determination of processor attributes eliminates the need for these devices, or at least reduces their size and complexity. Suppose one uses a processor to construct an intelligent network controller for a 10M-bit/second local area network adapter. The adapter will provide for all levels of protocol support and should be designed with a minimum system cost. The elimination of all extra hardware is highly desirable.

Hardware assistance requirements. FIFOs are in common usage in communication systems, primarily to overcome latency mismatches between devices. At 10M bits/s, only a 32-bit processor may provide the bandwidth necessary for continuous data transfer and still have enough remaining bandwidth for protocol algorithms. The alternative is to design an isolated buffer memory so that network data does not appear directly on the processor bus, thus allowing the program to execute during data transfers. Unfortunately, this design also adds extra components as separate memory and requires isolation buffers. One obtains the simplest and lowest cost interface when the data appears directly on the memory bus only once and the processor provides the maximum processing.

With a 32-bit-wide bus, a 10M-bit/s data stream requires the transfer of a memory word every 3.2 μs on the average. Suppose that the network protocol uses a packet format and a multidrop media. Each node is required to recognize at least three 32-bit, node-address values: local, group, and broadcast (all 1's address). This node should ignore any packet with an address not matching one of these address values.

With a 6.08-μs latency, the MC68020 would need at least two words of FIFO plus a shift register, delimiter detect circuitry, and a cyclic-redundancy-code (CRC) generator to interface to the line. If the processor is going to perform the node-address comparison itself, then it would have to reference these values from memory. This last step would require the addition of one more stage of FIFO to service the interrupting task. This stage holds data while the comparisons are made and the address pointers set up. Once the transfer is set up, then the processor could sustain the transfer under programmed I/O. However, the designer must be willing to suffer performance loss due to the significant interrupt overhead of the MC68020. The overhead amounts to 33 out of each 80 clock cycles—approximately 41 percent—plus the amount of time required to perform the transfer. A polling method of data transfer (the processor waits for data to become valid without returning from the exception) requires 100 percent of the bandwidth during network activity.

```
; Fast interrupt handler code, emulates a DMA controller by transferring data from the
; data FIFO to a memory buffer.  The critical path requires 5N + 9S cycles

            ORG     000001CH        ; FIRQ receiver interrupt handler
FIRQHAND    LDR     R8, [R10]       ; read data from FIFO
            TST     R9, #1          ; check for first transfer
            BEQ     TRANSFER        ; go to transfer routine if bit is zero
            CMR     R11, R10        ; compare for node address match
            CMRNE   R12, R10        ; compare for group address match
            CMPNE   R10, #FFH       ; compare for global address match
            OREQ    R9, #1          ; for match, enable DMA cycles
            STREQ   R10, [R9,LSR #6] ; write address into DMA buffer
            ADD     R9, #20H        ; update DMA address pointer
            MOVS    R15, R14        ; return

TRANSFER    STR     R8, [R9, #4]!   ; write data into DMA buffer
            SUB     R13, #1         ; update word count
            MOVNES  R15, R14        ; return

REOVRUN                             ; receiver overrun handler
```

Figure 6. Fast-interrupt handler routine.

If transfer overhead is too great or the amount of required processing is larger than that available to the interrupt service routine, the designer can add a DMAC device to the system to eliminate the overhead. Unfortunately, many DMAC devices cost as much as their respective processors. A second, unpleasant by-product of the DMAC device is that the processor is subsequently unavailable to process frame-level protocol. The designer must add another large amount of intelligent logic to deal with this protocol. Most MPU families have large-scale integration controllers to handle data communications protocols that are, again, just as expensive as the processor.

Using the VL86C010 can help minimize the amount of cost and board space required for intelligent data communications controllers. With a maximum latency of 1.87 μs and a fixed overhead of two processor cycles, the processor requires fewer FIFO stages and deals with frame-level protocol via software because much more processing time is available than the MC68020 provides. Line-interface support consists of a simple two-word FIFO, parallel-to-serial converter, serial-to-parallel converter, flag-detect comparator, and CRC generator. The processor performs all other functions, such as node-address recognition and data transfer. Figure 5 depicts how the FIRQ registers might support network framing protocol. This diagram assumes that a single FIRQ device in the system owns the FIRQ registers and a device prioritization is unnecessary. Figure 6 partially lists FIRQ handler software. The DMA handler requires 19 clock cycles (five nonsequential and nine sequential processor cycles) to process each transfer without an error. Please note that the VL86C010 requires only three processor cycles to compare the node addresses using the conditional execution.

Many DMAC devices cost as much as their respective processors.

Eliminating the DMAC device is extremely important, if possible: Almost all DMAC logic is redundant to that available on the processor. DMAC register values for transfer length and address are the only nonredundant logic; the processor has logic to increment address pointers and to interface to the memory bus. Designers place DMAC devices in the system solely to overcome the processor's inability to change context efficiently. Much of the logic is duplicated on the processor and the DMAC, increasing the cost and complexity of the system. The only reason DMAC devices function is that they do not have a large overhead at the beginning of each transfer.

Whenever designers implement an embedded controller, they should be aware that other characteristics besides performance dictate a wise choice of processor. Using a processor with the proper set of parameters—not just performance—minimizes hardware requirements and provides as much or more data throughput. For most controller situations, slight architectural modifications provide huge performance gains. The VL86C010 is an example of how careful analysis of the environment can yield a processor with outstanding performance and real-time I/O interface at a reasonable cost.

References

1. S.B. Furber and A.R. Wilson, "The Acorn RISC Machine—an Architectural View," *IEE Electronics & Power*, June 1987, pp. 402-405.
2. *MC68020 32-Bit Microprocessor User's Manual*, 2nd ed., Prentice-Hall, Inc., Englewood Cliffs, N.J., 1985.
3. W.A. Wulf, "Compilers and Computer Architecture," *Computer*, July 1981, pp. 41-47.
4. J.L. Hennessey, "VLSI Processor Architecture," *IEEE Trans. Computers*, Dec. 1984, pp. 1221-1246.

Ron Cates is a senior applications engineer in the Application Specific Logic Products Division at VLSI Technology, Inc. He has an extensive background in advanced computer architecture design and is currently responsible for system-level applications processor support. His current interests are controller design/definition and instruction-set definition for HLL support.

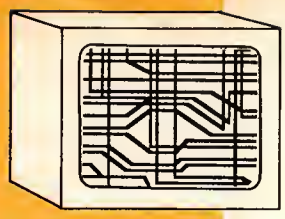
Cates received a BSEE from the University of Texas at Arlington and an MBA in management from Arizona State University.

Questions regarding this article may be directed to the author at VLSI Technology, Inc., 8375 South River Parkway, Tempe, AZ 85284.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 159 **Medium** 160 **High** 161



FEATURE

The TMS34010: An Embedded Microprocessor

Aimed at graphics systems, this 32-bit processor's large address reach, bit-field processing, and DRAM interface make it suitable for many other embedded processing applications.

Karl M. Guttag, Thomas M. Albers, Michael D. Asal, and Kevin G. Rose

Texas Instruments

The TMS34010 is a high-performance 32-bit microprocessor with special instructions and hardware for handling the bit-field data and address manipulations often associated with computer graphics. With integrated control and addressing for dynamic random access memory (DRAM), it supports a lower system cost than would normally be associated with a 32-bit microprocessor. Internal features such as an instruction cache, thirty-one 32-bit registers, and an independent memory control unit maintain a high degree of parallelism while efficiently utilizing lower cost DRAM.

Embedded processing for graphics was the target application for the 34010 from its inception. This led to a set of cost and performance decisions that are applicable to a wide variety of systems in addition to graphics systems.

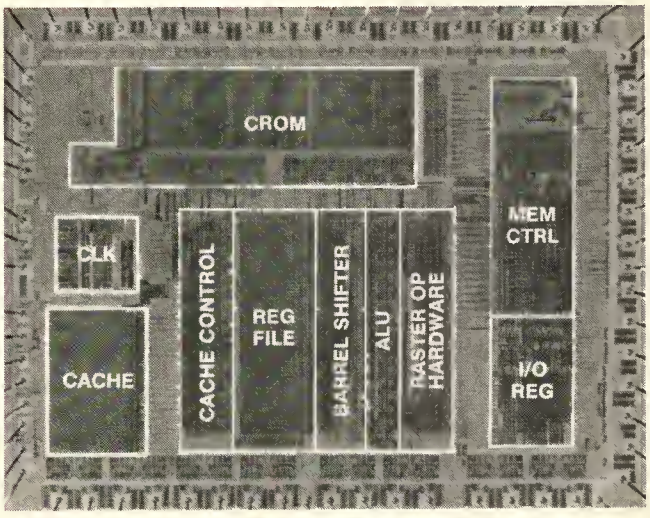
The chip contains over 180,000 transistors fabricated in 1.8-μm CMOS, consuming approximately one-half watt while executing in excess of six million instructions per second. In addition to a full 32-bit microprocessor core, the 34010 contains on-chip video random access memory (VRAM) display support, DRAM control, and a host interface.

History

Embedded microprocessors grew out of the need in the late 1960's for more advanced calculators.[1] Calculator designers recognized that as the variety and complexity of applications for calculators grew, programmable rather than fixed-function processors would be advantageous. Furthermore, engineers at Texas Instruments and Intel both recognized that these processors, once designed, could be used for much more than just calculators. The same chips could be used as general-purpose controllers, replacing gears, tubes, and relays with solid-state control. Texas Instruments initially pursued the single-chip microcomputer with its TMS1000 line, while Intel focused on the multichip microprocessor market with the 4004 and then the 8008.

Similarly, in the late 1970's and early 1980's designers began studying the processing needs of more advanced applications (such as digital signal processing, local area networks, and graphics). The comprehensive nature of these applications led designers at our company to the conclusion that even application-specific processors should be generally pro-

072-1732/88/0600-0039$01.00 © 1988 IEEE



Photomicrograph of the TMS34010 embedded microprocessor.

grammable. The design decision to support a completely general-purpose instruction set on an application-specific device means that the device will be well suited for many new application areas as well.

In display graphics, for example, the demands on graphics subsystems are growing rapidly. As desktop systems advance, a wide range of both graphic and nongraphic functions are being required of the graphics subsystem. The advent of graphical user interfaces for bitmapped graphics systems and the growing complexity of the interfaces between the system processor and its graphics subsystem dictate that the embedded graphics device should be programmable.

With advancements in processor technology, embedded control has expanded considerably. As many applications needed and could afford more processing power, 8-bit and, later, 16-bit CPUs came into use for embedded control applications. Today, many embedded applications are migrating to 32-bit microprocessors for their speed and advanced feature sets.

A 32-bit microprocessor offers significant advantages over older 8-bit and 16-bit chips, both in speed and ease of use. It has a large linear address reach for larger program and data requirements. The linear approach simplifies address management and tool requirements, and the larger address reach extends the application limits of the device. Additionally, 32-bit processors generally have much better bit-field processing capabilities than the older processors. Many 32-bit machines have internal instruction and/or data caches for faster program execution.

With the growing size and complexity of applications, the need for high-level-language (HLL) support has increased. However, for speed-critical portions of an application, the need for strong assembly language support for embedded control systems still exists. Thus, a processor that can be programmed easily at both levels is beneficial.

Embedded processors

The term *embedded processor* covers a wide range of processors and their applications. The term *embedded* implies that the user is not aware of the presence of the processor. That is, the user does not directly interact with or program the processor, but rather the processor provides a convenient method of implementing the control function. Microwave controllers, electronic games, and computer peripherals are typical examples of systems using embedded processors. With the falling cost and increasing power of microprocessor systems, the range and capabilities of embedded processors are expanding.

As 32-bit devices with their improved software support and speed move into the embedded control arena, it is natural to see them applied to the same applications currently supported by 8- and 16-bit devices. This trend tends to "raise the intelligence" of control systems while maintaining their embedded nature.

The evolution of printer systems clearly shows how the nature of embedded processors can change. The simple impact-head printers were implemented with 8-bit microcomputers as embedded processors. Today, high-resolution laser printers are emulating the impact printers they replaced and are providing the added capability of interpreting high-level page description languages. Consequently, the processing system of the laser printer is often more powerful than the host system it is connected to. Although the printer and its software interface have become more advanced, the processor is still being used in an embedded fashion.

In general, embedded systems are more cost sensitive than host systems, and large-volume embedded-system applications require relatively few chips. In embedded applications the processing is more a means to an end than an end in itself as in host applications. The embedded-system designer's goal is to provide the level of processing power necessary for the application with the highest system reliability at the lowest cost.

Applying host processors to embedded-processor systems. Most 32-bit microprocessor design to date has focused on host systems. These designs assume large systems such as multitiered memory systems with virtual demand paging memory management. Most of the new RISC (reduced instruction set computer) machines also require specialized memory systems such as fast memory subsystems (in some cases special external caches), very wide buses, and sometimes multiple buses. Another important factor is the bus speeds that many of these new processors require; these speeds are beyond most available application-specific ICs (ASICs) and can mean requiring very fast external logic with relatively low levels of integration.

These features are desirable for larger host applications, but for most embedded applications they are either unnecessary or too expensive. The new pro-

cessors, for example, typically require ceramic packages of 100 pins or more, resulting in higher component and system costs.

Thus, a large practical gap in system complexity and cost lies between most 32-bit microprocessors and their 16- and 8-bit predecessors. The needs of embedded controller applications for faster and easier-to-use microprocessors without the unnecessary impediments associated with host systems are not being addressed by most 32-bit processors.

Figure 1 diagrams the trade-offs in the current microprocessor market. On one axis is the relative system cost of the processor and its intended memory system, and on the other axis is relative performance. The general-purpose 32-bit microprocessors lie in the high-performance and high-system-cost region of the graph. The processors alone for these systems cost over $300 and require fast memory in the form of external static RAM (SRAM) caches to achieve optimum performance. The RISC chips on today's commercial market likewise have been designed for high-performance and high-cost systems. At the lower cost and lower performance end are the 8-bit and 16-bit microprocessors and microcomputers. In the case of microcomputers the memory is built into the processor chip. The 8- and 16-bit microprocessors typically have SRAM interfaces and require external logic to connect to lower cost DRAM. Consequently, high-cost SRAM has been used for lower chip count memory systems.

The 34010 is positioned between the two extremes in the cost-performance trade-off. It offers substantially more performance than the 16-bit microprocessors and lower system cost than the 32-bit general-purpose processors. The two positions of the 34010 on the graph in Figure 1 depict its relative performance in general-purpose processing and its advantage in graphics due to its special graphics processing hardware.

Application microprocessors. An application microprocessor is a general-purpose microprocessor designed with special hardware and instruction support for a specific system application. These microprocessors provide significant performance and cost advantages for applications needing their special capabilities. Application microprocessors grew out of the recognition that certain functions occur more frequently in embedded applications than in general-purpose processing.

The TMS320 digital signal processor (DSP) is a good example of an application processor family aimed at a specific application area. Designed for DSP applications, it optimizes the performance of high-speed multiplication operations typical in these applications. Many other applications have made use of the unique capabilities of DSP chips, including such varied applications as voice recognition, adaptive suspension, and image compression for graphics.

The TMS340 graphics system processor (GSP) family, of which the 34010 is the first processor, is aimed at

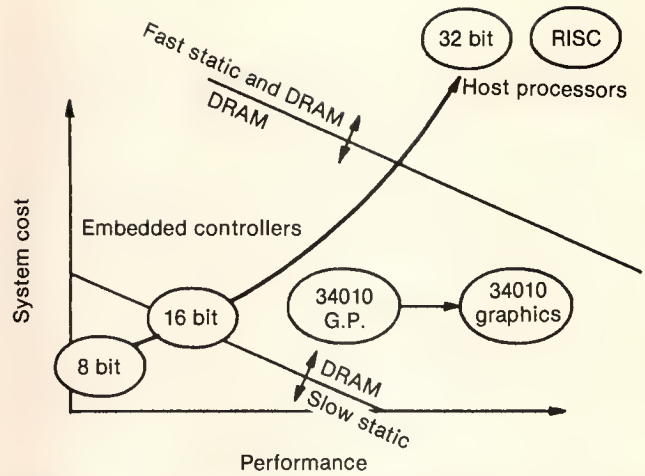


Figure 1. Embedded controller system trade-offs.

the bit-field processing and large memory spaces associated with graphics rendering. Its large address reach, bit-field processing capability, on-chip timers, and DRAM interface make the processor well suited to many embedded-processing applications.

Focus on embedded processing. Because it was intended to be an embedded processor or second processor in many systems, the 34010 has a set of features focused on reducing system complexity. It assumes a small system model with a single external memory hierarchy. The silicon budget was put into such features as DRAM control, timers, bit-field processing, pixel processing, and simple connection to a host processor or communication channel.

Figure 2 shows the distribution of the processor's 68 pins. Table 1 lists the functions of these pins. Note the

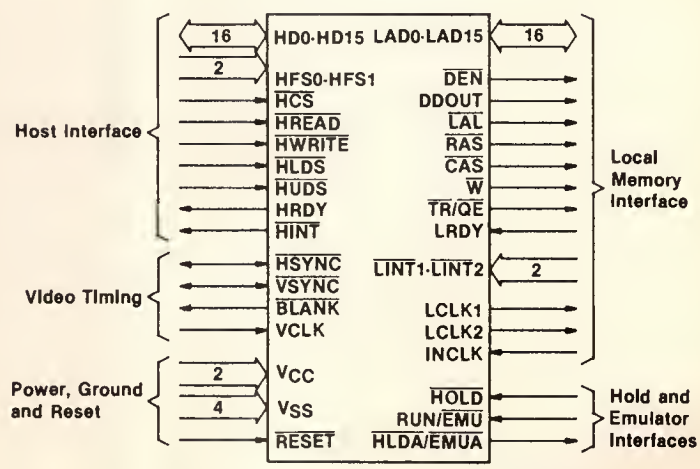


Figure 2. TMS34010 pin diagram.

Table 1. TMS34010 pin description.

Name	Pin	I/O	Description
Host Interface Bus Pins			
$\overline{HCS}$	66	I	Host chip select
HD0–HD15	44–51,53–60	I/O	Host bidirectional data bus
HFS0,HFS1	67,68	I	Host function select
$\overline{HINT}$	42	O	Host interrupt request
$\overline{HLDS}$	63	I	Host lower data select
$\overline{HUDS}$	62	I	Host upper data select
HRDY	43	O	Host ready
$\overline{HREAD}$	64	I	Host read strobe
$\overline{HWRITE}$	65	I	Host write strobe
Local Interface Bus Pins			
$\overline{RAS}$	38	O	Local row-address strobe
$\overline{CAS}$	39	O	Local column-address strobe
DDOUT	36	O	Local data direction out
$\overline{DEN}$	37	O	Local data enable
LAD0–LAD15	10–17,19–26	I/O	Local address/data bus
$\overline{LAL}$	34	O	Local address latched
LCLK1,LCLK2	28,29	O	Local output clocks
$\overline{LINT1}$,$\overline{LINT2}$	6,7	I	Local interrupt request pins
LRDY	9	I	Local ready
$\overline{TR/QE}$	41	O	Local shift-register transfer or output enable
$\overline{W}$	40	O	Local write strobe
INCLK	5	I	Input clock
Hold and Emulation			
$\overline{HOLD}$	8	I	Hold request
RUN/$\overline{EMU}$	2	I	Run/Emulate
$\overline{HLDA/EMUA}$	33	O	Hold acknowledge or emulate acknowledge
Video Timing Signals			
$\overline{BLANK}$	32	O	Blanking
$\overline{HSYNC}$	30	I/O	Horizontal sync
VCLK	4	I	Video clock
$\overline{VSYNC}$	31	I/O	Vertical sync
Miscellaneous			
$\overline{RESET}$	3	I	Device reset
V_{CC}	27,61	I	Nominal 5-volt power supply
V_{SS}	1,18,35,52	I	Ground

emphasis both on adequate local bus control and on a host interface for attached processing. More than one third of the pins on the device are dedicated to this embedded function support. Almost half the pins support the local address/data bus designed to make DRAM interfacing straightforward. The device performs the row/column address multiplexing necessary to interface efficiently to DRAM.

Overview of the internal architecture

As shown in Figure 3, the internal architecture of the TMS34010 consists of six major blocks: main CPU, instruction cache, memory controller, host interface, display controller, and internal clocks.[2-4]

Main CPU. The CPU is controlled by an 808 microstate control ROM (CROM). It can execute simple instructions in a single cycle when in cache, and it can perform complex microsequences such as the pixel block transfer instructions (PIXBLTs). Each CROM word has 166 bits, which support highly parallel operations within the CPU.

The main internal data paths are 32 bits wide and contain the key elements for efficient execution of both graphics and general-purpose instructions. The thirty-one 32-bit registers are organized as two register files, each with 15 registers sharing a common stack pointer register. The ALU, adder/subtracter, and barrel shifter have separate inputs and control and can all operate in parallel.

Instruction cache. The instruction cache, transparent to the programmer, was designed to support fast execution while using DRAM for the system memory. During the execution of time-critical loops, the cache helps in two ways: It supports fast instruction fetches, and it frees the memory bus for reading and writing. The programming model is a single memory space for instructions and data, and the cache is used to separate them for parallel access.

The 256-byte cache uses a four-way set associative with four segments (sets), eight subsegments per segment, and four words per subsegment. Each subsegment has a "present" bit, and direct replacement of "misses" within a subsegment is made for the four words. The four segments use a four-location CAM (content-addressable memory) to determine whether a segment is present and a four-location LRU (least recently used) stack to determine which segment is replaced on a miss.

Memory controller. The memory controller is actually a separate microcoded processor with its own control ROM that coordinates all accesses to local memory. CPU, host, and video requests are prioritized and scheduled by the memory controller along with DRAM refresh cycles. The memory controller is responsible for generating the control signals for the local memory bus.

All the necessary DRAM and VRAM control signals are generated by the memory controller. The row and column addresses along with data are triple-multiplexed on the 16-bit local address/data (LAD) bus under control of the memory controller. Only one external buffer/latch is required to connect the processor to DRAM.

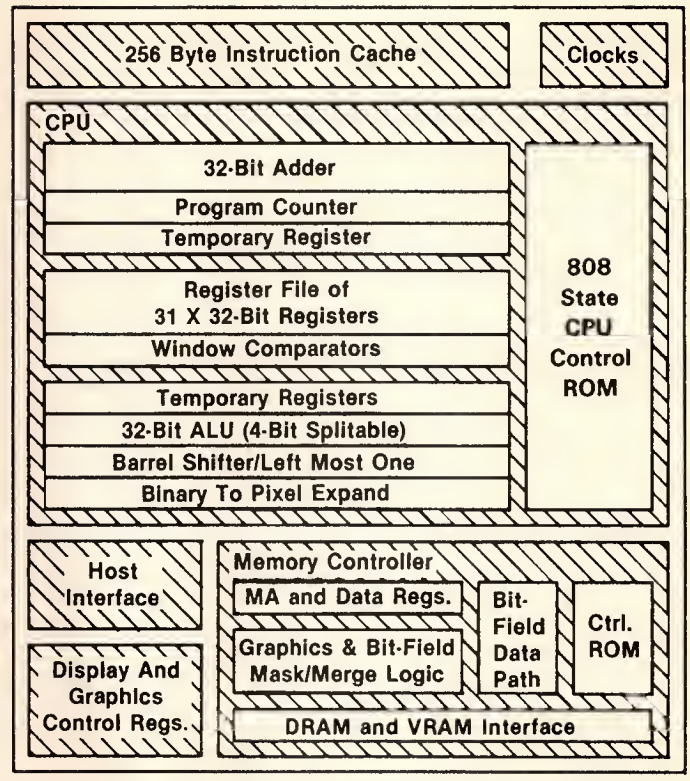


Figure 3. TMS34010 internal architecture.

Figure 4 shows the local bus timing. The signals LCLK1 and LCLK2 are the local bus clocks generated by the processor for timing on the bus. The $\overline{RAS}$ (row address strobe) and $\overline{CAS}$ (column address strobe) signals directly generate the timing required by DRAM. The $\overline{LAL}$ (local address latch) signal, which falls after the column address is valid, is used to latch the column address. For static memory interfacing, the $\overline{RAS}$ signal latches the upper address bits, and the $\overline{LAL}$ signal latches the lower address bits. $\overline{W}$ (write enable) is designed to give the proper write signal for DRAM. The $\overline{DEN}$ (data enable) and DDOUT (data direction out) signals enable and control the direction of bus transceivers if necessary in the system. $\overline{LRDY}$ (local ready) is a processor input used to lengthen memory cycles for slower memories and peripherals.

The $\overline{TR}/\overline{QE}$ (shift register transfer/output enable) signal, controlling a corresponding input on the VRAM, serves two dissimilar functions. It initiates the shift register transfer cycle on the VRAM, and it is timed to control the output enables on the VRAM and the 4-bit-wide DRAM devices.

The mixing of the dissimilar functions of the VRAM's shift register transfer signal and its output enable on $\overline{TR}/\overline{QE}$ (alternatively named $\overline{TR}/\overline{G}$) came about because the 34010 was defined in conjunction with the original VRAM, the TMS4161.[5] The purpose of the output enable function was to make it un-

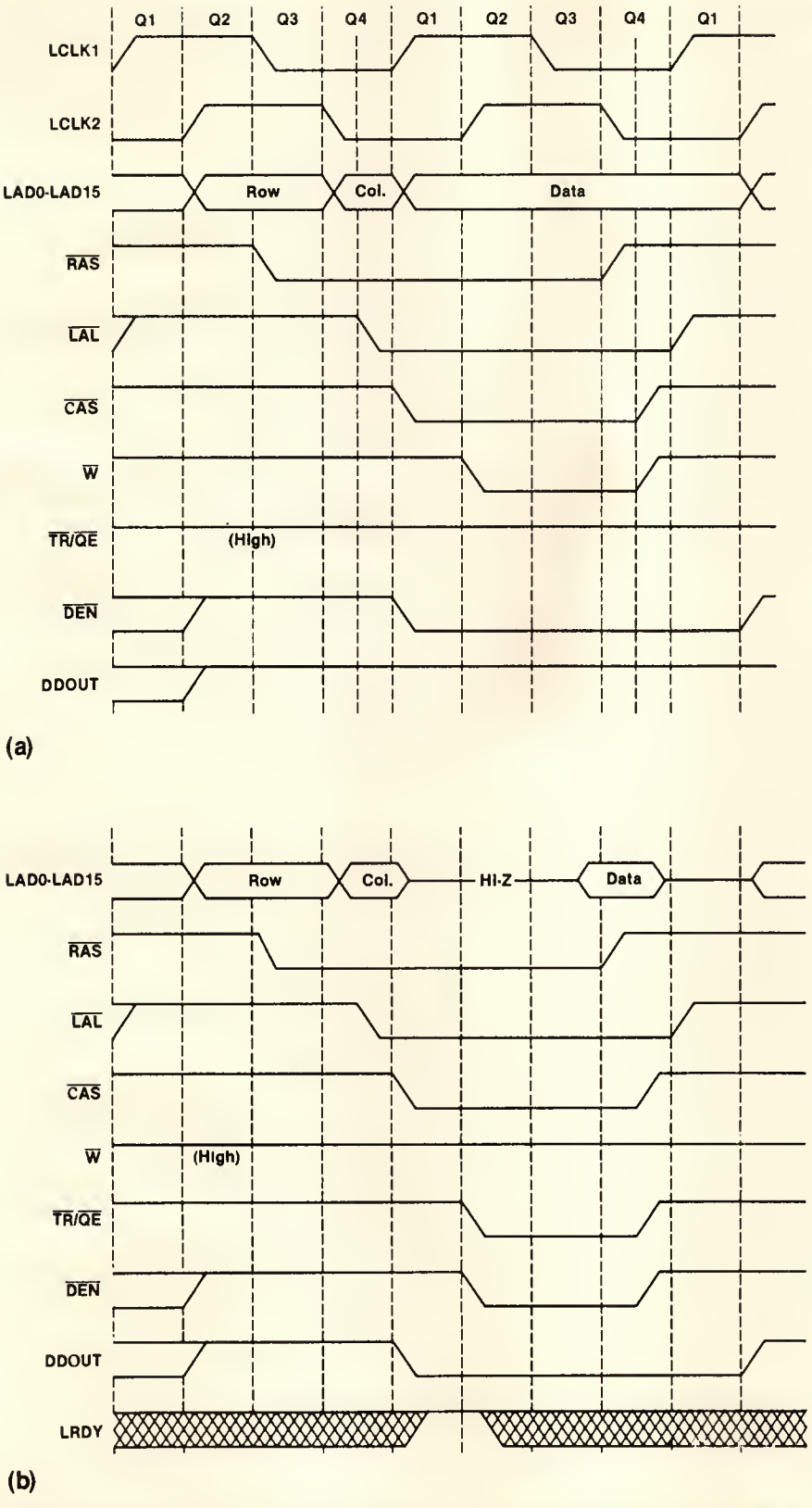


Figure 4. Local bus cycle timing: (a) write cycle; (b) read cycle.

necessary to add extra buffers between the VRAM and the 34010, but there were no pins left on the first VRAM. The $\overline{TR}$ signal already existed, so the designers decided to time-multiplex the signal. This invention of necessity has become standard on all VRAMs designed since.

The memory controller also plays an important role in offloading the main CPU from the burden of bit-field processing. On bit-field operations (including any move instructions), the CPU simply passes the starting bit address, the field size, and the data to the memory controller; then the CPU is free to execute the next instruction out of cache. Before sending the data, the CPU uses its barrel shifter to get the data in the proper bit alignment, but the memory controller actually performs the masking and merging operations to insert the field. On the basis of field size and address alignment, it computes and schedules as many read and write cycles as necessary, requiring no further interaction with the main CPU.

Host interface. Unlike other microprocessors, the 34010 has a dedicated interface port to allow another processor to gain access to its memory. The host interface is actually a communication channel into the 34010's memory space and can be used for other purposes. Accesses requested via this 8- or 16-bit data interface are scheduled at higher priority than the 34010's CPU by the memory controller. The local memory can also be directly accessed by a conventional hold interface.

Four internal memory-mapped registers are dedicated to the host. These registers are loaded from the 8/16-bit host bus under the control of two function select pins (HFS0, HFS1). Two of the 16-bit registers combine to form a 32-bit address into local memory. Another register holds the data written to and

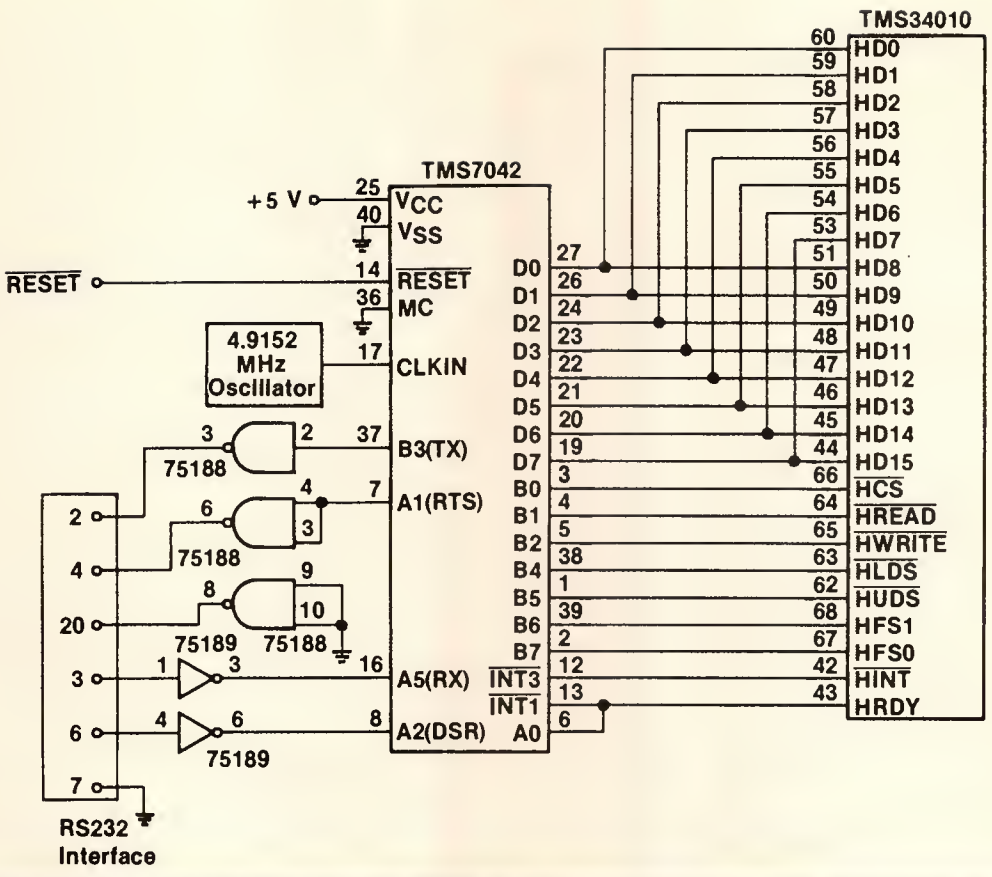


Figure 5. TMS7042 as serial port host.

read from memory, and the fourth contains control information.

In addition to the data transfer registers, the host processor has access to a 16-bit control word. Using this register, the host has complete control of the 34010. The control register can be set up for automatic incrementing of the address register on reads and/or writes for block access throughput of five megabytes per second.

The management of these resources and the memory pointer registers can be controlled by the 34010 or the host. In a 34010-controlled interface scheme, the host does not need to have any knowledge of the local memory organization. The 34010 is capable of loading the address registers locally, thus decoupling the host from the local memory implementation. The host interface's control register provides the host access to dedicated message-passing bits, a register-controlled interrupt in and out, and a nonmaskable interrupt. The control register also supports halting of the 34010's CPU and flushing of the contents cache, particularly useful when downloading code. The halt control can also be very useful in capturing the full bandwidth of the local bus for time-critical data transfers.

Using the host interface for other functions. Although intended to be a port for a host processor's commands and data, the host interface provides an economical way to access the local memory for any purpose. One system, for example, uses an 8-bit microcomputer as a front-end serial controller. The microcomputer, operating as a serial port or network interface, attaches very easily to the 34010. Figure 5 shows the connection of the 8-bit TMS7042 microcomputer with serial port to the processor host port.

The host has access to the interrupt vector table via the host interface. Thus, the host or other local processor can dynamically install interrupt vectors and their associated interrupt routines. The external processor can then initiate interrupts via either the 34010's interrupt pins or the host control register. For externally generated interrupts, there are both maskable and nonmaskable interrupts in addition to Reset. Two external pins, LINT1 and LINT2, are dedicated external interrupt inputs. Two bits in the host control register control the operation of the nonmaskable interrupt. One bit invokes the interrupt itself while the other enables or disables the stacking of the program counter and status during the interrupt routine. Preventing the

stacking process in the interrupt routine is sometimes necessary to gain immediate control of a system in which the stack pointer value may have been corrupted.

The host interface can also be used as an independent debug/test channel into the local memory space. Debugging programs can use this port to access state variables supplied by a local monitor program giving information about the processor's internal machine state. Similarly, the port can be used for communicating system state variables at the end of prescribed system tests at bootup. These can be either controlled locally by the 34010 or command-driven over the host port.

Display controller. Although designed for CRT control, the display controller is a very flexible counter that can be used for a wide range of timing or event-counting functions. Functionally, it has two cascaded 16-bit counters (for a total of 32 bits of dynamic range) with four programmable comparators on each counter. The comparators are used to generate three output signals (nominally used as vertical sync, horizontal sync, and blanking). A programmable interrupt can be set on the basis of any vertical count.

The input clock for the counter can run from 0 (stopped) to 7.5 MHz and can be totally asynchronous with the processor clock. The counters support an external video mode, which allows external events to reset the vertical and horizontal counters independently.

Internal clocks. The clock timing logic converts the input clock frequency into the various internal timing clocks needed to operate the processor. In addition, it generates the local bus clock signals used by external devices to operate synchronously with the processor's local bus. Current devices operate with a divide-by-eight from the input clock to generate 130-ns cycle times for a 60-MHz input clock.

Model of operation. The large register file, instruction cache, and independent memory controller are designed to work together for high performance. The processor's model of operation is that instructions controlling the algorithm are automatically loaded into the cache, data is stored in the large register file, and the memory controller and its bandwidth into the off-chip memory are used only in manipulating external data. Other system operations such as host accesses are requested and scheduled by the memory controller as a background task, and interrupts are used for communications, handshaking, and error conditions.

Feature set and definition decisions

In designing a processor, one must evaluate package size, pin count, target memory type, and a wide variety of features that can be incorporated. The 34010 was designed to bring the high performance and ease of programming associated with 32-bit microprocessors to low-cost systems. Contrary to common belief, the difficult part of product definition is not identifying good features to add (which are infinite) but making the tough choices between what can be included and what must be left out for cost reasons.

To achieve the best system cost-performance ratio, the feature set and functions of a processor must be balanced. *Balanced* means that the features complement each other in a practical way. For example, the 34010 was targeted at low-cost memory systems. Features such as the on-chip instruction cache and large register file provide faster execution by reducing the need for access to the DRAM, while direct DRAM control and multiplexed addressing reduce system cost and complexity.

Large linear address space with bit-field processing. Graphics display systems need large amounts of memory, leading to several basic design decisions. A clean architecture to support a large linear address reach is needed, and this in turn requires a 32-bit internal data path to manipulate the large addresses quickly.

Unique among microprocessors, the 32-bit address of the 34010 points to the exact bit location in memory rather than to the byte, word, or long-word of other processors. The whole of memory is viewed as a series of bits ordered from 0 to $2^{32}-1$. All the memory addressing modes directly support bit-field processing. Field lengths from 1 to 32 bits are directly supported in all general-purpose move operations. The autodecrement and autoincrement addressing modes also use the field size to adjust address registers. In addition, any size array of fields can be moved with the PIXBLT instruction (pixel block transfer).

The byte, word, and long-word are artifacts of older processor architectures in which there was only one basic data size—the byte. In processors with limited address bits, byte addressing served as a good compromise between data granularity and address reach. Also, earlier architectures did not have the hardware, such as barrel shifters and mask/merge multiplexers, to quickly handle problems bit-aligned addressing can create. But the 34010's architecture started with a 32-bit address reach, and its hardware manipulates bit fields with equal ease as bytes or words, so there was no need to impose a distinction.

Bit-field processing is directly supported by special hardware on the device. This extends the bit control and manipulation facilities found in controller systems, adding memory bit manipulations to those available in registers. Processors without this facility must perform many operations to achieve the same effect. To transfer a nonaligned byte field from one memory location to another, the typical processor must read the source word into a register, mask out uninvolved bits, align the source word with the destination location,

Contrary to common belief, the difficult part of product definition is not identifying good features to add.

read in the destination word, mask out the affected field (byte), logically merge the source and destination words, and then write the result to the destination. The 34010 CPU can direct this operation within a single instruction and then continue to execute operations within its register file while the destination memory accesses are being performed.

Large register file. The decision to go to thirty-one 32-bit registers rather than the 16 or fewer found on most machines was driven by the desire to make time-critical functions run faster and to ease assembly-level programming. Register-to-register operations occur in a single cycle when running out of cache and can occur in parallel with the memory controller's completion of previously started write cycles. This parallelism naturally occurs in routines in which the CPU is computing functions written to a series of memory locations. The example used as a model during the 34010's definition was an ellipse-drawing routine, in which the address computations and data values are held in the register file and the pixels to be written are sent to the memory controller. A large register file means that all the parameters for most time-critical functions can be kept inside the processor, thus preventing the thrashing of parameters between the register file and memory. By preventing thrashing, the register file frees the memory bandwidth for other functions such as memory write cycles, host accesses, and DRAM refresh. In many programs the large register file can be the single most important feature for improving performance.

There are several ways to take advantage of the larger register file. It is helpful to the compiler to have several working registers for storing intermediate results of computations such as evaluating expression trees. Intelligent compilers (made popular by RISC architectures) can take even greater advantage of more registers by tracking the contents of registers and performing efficient constant generation. This also means that the programmer trying to optimize his or her own code in high-level language will not compete with the compiler for access to a small number of register variables to hold critical values.

Important, time-critical routines are usually written in assembly code to optimize performance. It is in writing such routines that the benefit of the large register file is most obvious. If there are too few registers, most of a programmer's algorithmic effort can be expended on register management. With 31 registers, a large number of critical values can be kept in the register file during execution. This is sufficient for most control algorithms and has a direct impact on the ease of design of the algorithm implementation. In addition, the processor's efficient register-stacking and -unstacking instructions make register allocation and management trivial.

The 34010's initial target applications area clearly indicated that a large number of 32-bit registers were very desirable. Many graphics algorithms require a large number of data variables and address pointers. Without the target application to measure against, it would have been much easier to define an architecture with fewer registers.

The on-board registers are organized as two 15-register files named the A and B files. The distinction between the files is that instructions requiring two registers must have both registers in the same file. The move register to register instruction is an exception to this rule so that data can be moved between register files. The other exception is the stack pointer that is accessible as the 16th register of either file.

The memory move, arithmetic, Boolean, shift, and other register-based instructions and addressing modes can use any of the 31 registers. This is particularly important in optimizing languages such as C, where any operation that can be performed on data can be done on address pointers. Therefore, data/address placement is not constrained, giving the compiler great latitude in organizing register usage.

An important part of the processing task for embedded-control applications is the processor's interrupt support. An added benefit of the large register file is the ability to *dedicate* registers to time-critical functions such as interrupt routines. Embedded applications often have parameters that must be dealt with quickly when an interrupt occurs. Dedicating part of the register file to these values can save considerable time swapping data in and out.

Instruction cache. The four-way set associative approach was designed to support two loops, each straddling set boundaries without thrashing. During the 34010's early definition, relatively little time was spent defining the cache compared with time spent justifying its hypothetical performance on pathological cases, cache architecture being part philosophy and part science.

Since graphics was the focus of the 34010, circle, line, and ellipse algorithms were used as model test cases for the cache. The nature of these algorithms is that they have a single loop, but a decision based on incremental calculations is made between two sets of code on each loop. The goal was to have these algorithms fit in the cache without thrashing and without requiring the programmer to worry about the position of code in memory. The four-way set associative method worked well on the graphics test cases.

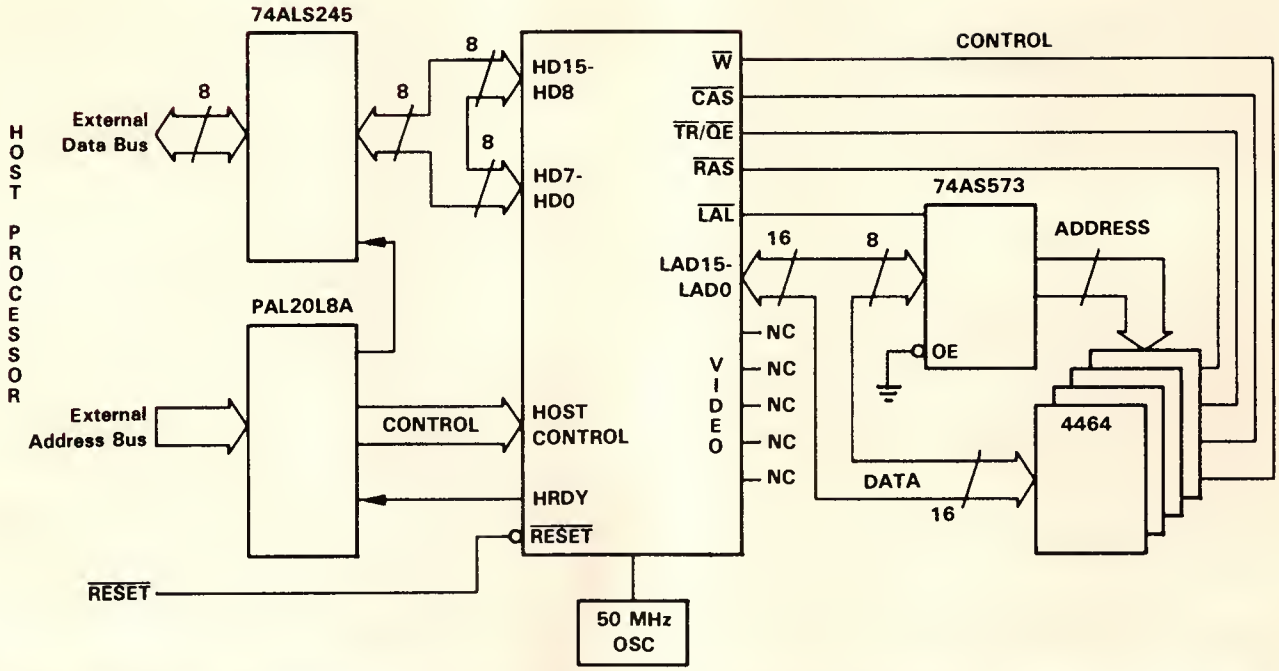


Figure 6. TMS34010 DRAM system.

Unlike general data caches, relatively small instruction caches can have very good hit rates. This is due to the locality of programs and the fact that many time-critical functions are done in small loops. Additionally, instruction caches are less expensive than general data caches since they do not have to deal with data writing back to main memory (unless self-modifying code is allowed).

Making DRAM accessible to low-chip-count systems. A key difference between the 34010 and other microprocessors is its direct interface to DRAM. This interface changes the point at which DRAM becomes cost effective in a system.

Figure 6 shows the minimum number of devices required to implement an embedded system in a low-cost PC platform. A PAL (programmable array logic) and an octal transceiver provide the host processor access to the 34010's five-chip, 128K-byte DRAM system. The host processor provides nonvolatile program store and bootstrap capability for the system. This provides two benefits. First, the host has random access into the processor's memory space for communications, eliminating the need for a separate I/O channel. Second, the operating software of the embedded processor can be updated or entirely changed remotely from the host system. This eliminates the need for exchanging parts or otherwise "touching" the embedded system hardware in order to perform field upgrades.

Although DRAMs are the most cost-effective memory device per bit, they have not generally been used in low-cost, low-chip-count systems. DRAM devices require complex timing and address multiplexing, which, if not built into the microprocessor, require external control. Because of this overhead, applications needing relatively few memory chips have not previously used DRAM.

The capacity of DRAM chips allows the designer fairly large amounts of memory in very few chips. With 256K-bit DRAMs organized as 64K deep by 4 bits wide per chip, only four chips give a 16-bit data width and 128K bytes of memory. With 256K by 4 (one megabit) DRAM devices, four chips result in 512K bytes. Thus, a relatively small number of DRAM devices can provide all the RAM required for most embedded applications.

Having the DRAM interface built into the processor can provide performance benefits in systems using these lower cost memories. Typically, considerable time is lost in interfacing a microprocessor to the DRAM because of the inherent losses in going through another controller chip and because of timing mismatches between the microprocessor's signals and the DRAM. The processor uses a high-frequency (40 MHz to 60 MHz) input clock so that it can precisely place the large number of timing edges required by a DRAM.

Since DRAM is generally slower than other memory types, considerable attention was given to improving

performance in a DRAM-based system. While "big system" host models may use multitiered memory hierarchies with external instruction and data caches and very fast buses to achieve very high performance, these features come at a cost. The instruction cache and the large register file were designed to reduce accesses to the DRAM.

Packaging for low cost. The processor requires only 68 pins and power dissipation of about one-half watt, allowing it to use a very low cost plastic package. Packaging dominates the cost of making most high-volume components, so fitting a low-cost package was an important design consideration. One key to keeping the pin count small lay in the DRAM timing. Time multiplexing the row address, column address, and data on the same pins fit the DRAM timing requirements well and saved a number of pins.

While the 34010 has a 32-bit internal data path, the external data path is 16 bits to reduce overall system cost. This design decision kept the pin count down and eliminated the overhead associated with a wider data bus. As part of the original strategy, wider data bus versions of the device are under development.

Opcode formats and design simplicity. The 34010 follows the RISC philosophy of having very few fixed opcode formats and has a fixed-length 16-bit instruction. As other researchers have indicated, a 32-bit opcode, as in most RISC machines, is not efficient in terms of the number of bits required by a function.[6] The same philosophy of architectural efficiency has resulted many times in inefficient instruction encoding. Opcodes (not including immediate data) on the 34010 were kept to 16 bits in length to reduce instruction bandwidth requirements and to obtain better utilization of the instruction cache.

Another objective was to make the instruction formats easy to decode. There are only four opcode formats: one-register, two-register, short-constant, and jump opcodes, as shown in Figure 7. These four formats are organized into two groups for decoding purposes: single-register and everything else. For the one-register format a special nonbinary address decoder was used to avoid an extra level of decoding. Thus, one less level of instruction pipelining was required, resulting in simpler logic and faster jumps and branch execution. For the other formats, the upper 8 bits (bits 0 to 15) of the opcode specify the instruction. Redundant states in the microcode preclude the need for detailed decoding beyond the upper 8 bits.

The two-register-file organization was devised to provide a large register file and at the same time to maintain a compact, easy-to-decode, 16-bit instruction word. This organization requires only 9 bits (since the register file select bit is shared) as opposed to 10 bits to specify two of 31 different registers, but it also limits operations between files. Other approaches were considered, such as split address/data files as was done on the 68000.[7] But that organization might have restricted the operations that could be done on addresses, limited the flexibility of register usage for address or data, and made instruction decoding more complicated.

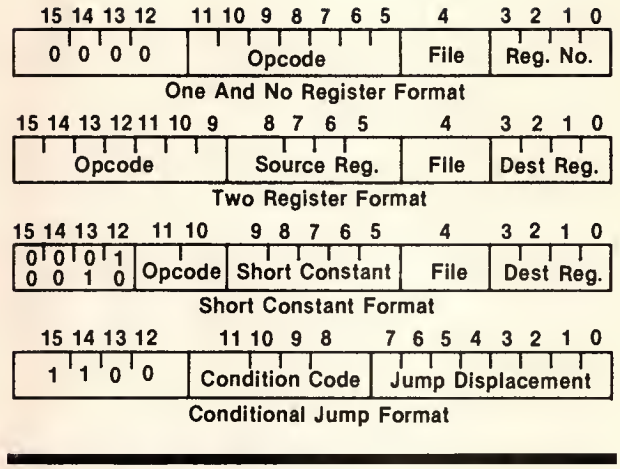


Figure 7. TMS34010 opcode formats.

RISC. What discussion of microprocessor architecture decisions would be complete today without commenting on RISCs, a much overused and abused term?[8] The 34010 design, influenced by the Berkeley RISC philosophy, has an RISC base instruction set to which were added special graphics instructions. Independent of the RISC influence, our experience designing the 9900 family of microprocessors had demonstrated that decoding complex instructions wastes hardware and performance.

Statistical data on general applications running on our 990 minicomputers agreed with other studies indicating that move, jump, increment, decrement, and add operations dominate the mix of instructions executed.[9,10] In contrast, embedded applications can stress a specific function and thus may have instruction mixes that look very dissimilar to the general-purpose cases.

The 34010 supports variable bit-field sizes, not found in many RISCs, but it does adopt the Berkeley RISC concept of sign or zero extending to 32 bits for smaller data types when moving them into a register.[11] Like the Berkeley RISC, it performs 32-bit register-to-register arithmetic.

Although it does not use the Berkeley register window concept, the 34010 does offer a larger register file than most 32-bit microprocessors. Some consideration was given to the register window concept since it is similar to the older workspace pointer concept of the TMS9900 family, which used a pointer to its register file in external RAM. The 9995 microprocessor had 256 bytes of internal memory, typically used to hold a large number of workspace "registers," but this was still in slower RAM rather than faster, dual-ported reg-

ister file storage. The 34010's designers decided that the very large windowed file of the Berkeley RISC would add too much to the chip's size and would create speed paths that might limit fast operation.

In keeping with the RISC philosophy, the 34010 executes most basic instructions in a single cycle. By using a single-level, very wide (166-bit) control word, the processor can execute single-cycle instructions without extra pipelining. The opcodes were kept very simple and were constructed to act directly as addresses into the microcoded ROM. This eliminated the extra decode logic normally associated with microcoded microprocessors. In effect, whereas a RISC machine uses a PLA for instruction decoding, a single-level ROM lookup is sufficient on the 34010.

Departing from the strict RISC philosophy, the designers recognized that applications would benefit greatly from more sophisticated instructions that do not fit the single-cycle model of pure RISC machines. The complex PIXBLT instructions can be pipelined much more efficiently as single instructions than if broken into multiple single-cycle instructions. With internal microcoding, the address manipulations, field extractions, merging, and multiple memory cycles can be more efficiently coordinated. The wide control supports many parallel operations for faster execution.

Real-time software development. The importance of assembly and HLL support was the basis for TI's decision to support the 34010 family with source and object management tools for assembly and C. In addition, both real-time and software-only emulation tools provide debugging capability for hardware and algorithm prototyping. Application libraries provide source code algorithms for specific design tasks such as CCITT (International Consultative Committee for Telegraphy and Telephony) Group 3/Group 4 compression and decompression, as well as for various graphics standards from the Massachusetts Institute of Technology, the American National Standards Institute, Graphic Software Systems, Microsoft, and others.[12] Embedded-processor applications also usually require real-time operating systems. These have been developed specifically for embedded processing for the 34010 and are available from external parties.[12,13]

Applications in embedded control

The general-purpose processing power and low system cost of the 34010 make it useful for a wide number of applications. Because of its graphics hardware and instructions, it naturally found wider initial use in graphics and graphics-related applications, but over time more designers are finding its larger applicability. The bit-field processing is an important feature that makes it attractive in control and data compression applications.

Graphics terminal and display systems. The 34010 has been widely used in both graphics add-in boards and display terminals.[12] The hardware support for video display terminals, DRAM, VRAM, and host interface greatly reduces the cost of these systems. The support for graphics drawing, X-Y addressing, pixel block transfers, and general-purpose processing provides high performance and easier programming.

The 34010 can also be used to offload nongraphic processing from the host to improve overall system performance. As a result, entire applications have been written to run directly on the embedded processor, using the host only for keyboard interface, disk drive, and other I/O functions.

Consumer electronics. Because it is both a powerful microprocessor and a graphics chip integrated in low-cost packaging, the 34010 has potential applications in new consumer graphics designs. To date, most video game systems have used a combination of an 8-bit microprocessor and a video controller chip to support graphics functions such as sprites (two-dimensional X-Y positional characters). The microprocessor and the controller chip can be replaced by a single 34010. The processing power of the resultant system opens up whole new areas of education as well as entertainment. Electronic building blocks, home 3-D CAD, flight simulators, driving trainers, and 3-D adventure games are just a few of the possibilities.

Image compression for facsimile and CD-ROM. Graphics and images require large amounts of data to be stored and/or transmitted, making data compression essential both for performance and cost. The need to globally transmit images has resulted in the CCITT facsimile standards for data compression.[14,15] Inexpensive stand-alone systems can be constructed incorporating CCITT Group 3, a 9600-bps modem, and paper-handling control. The acceptance of CCITT Group 3 compression has led to its wider use in other applications. A laser printer with the addition of a 9600-bps modem can double as a facsimile printer. The CCITT standard is also being used as a compression method in applications such as CD-ROM (compact disks) and scanners.

Fax-modem PC add-in boards provide facsimile transmission directly from a PC. They have a considerable quality advantage over printing out the image and then having it scanned by a stand-alone fax machine because they eliminate the distortion introduced during the scanning process. Since CCITT encoding is a generally accepted standard, it is also a convenient way to communicate images between machines without an intermediate step to paper.

The CCITT Group 3 and Group 4 standards are based on bit manipulations and variable field length encoding. This bit-field processing is applied to edge detection, run length encoding, and data movement. A graphics processor has the added benefit of being able

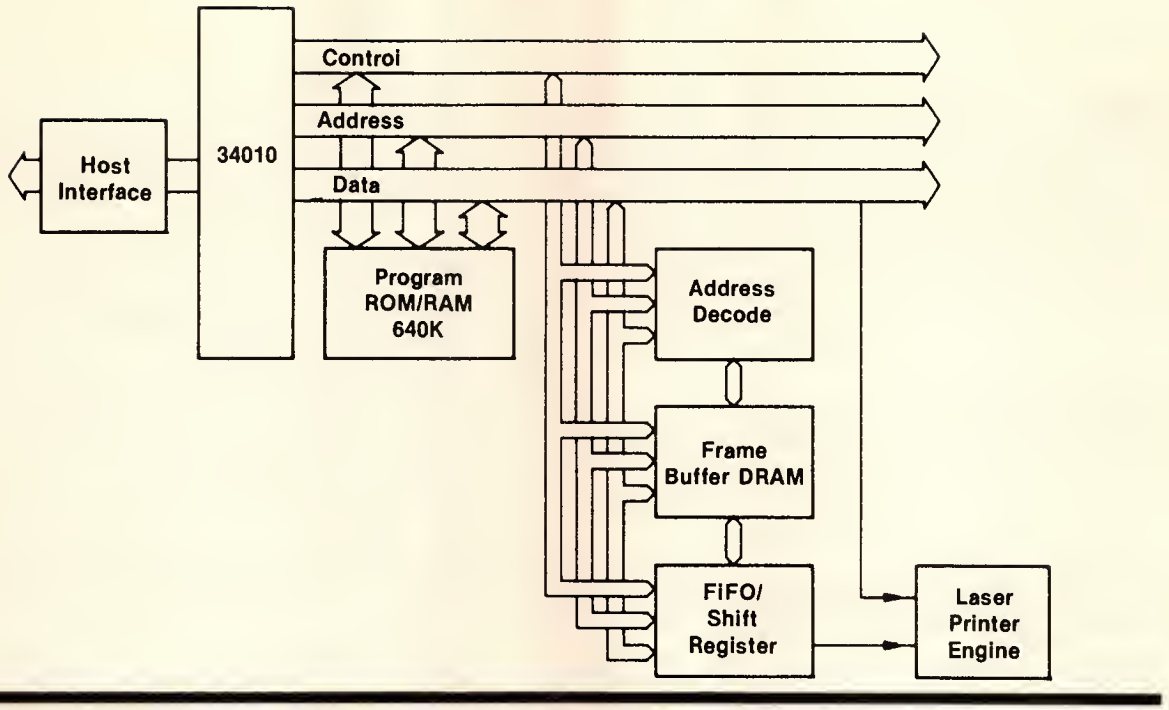


Figure 8. Laser printer system.

to generate, display, and manipulate the resulting images. In the fax-modem application, although most PC displays cannot display the high-resolution fax image, the image can be translated into a gray-scale image that can be displayed for preview.

Data compression techniques are being applied to increase the storage capabilities of such high-density media as CD-ROMs and other optical laser disks. As the storage requirements for these media increase, a higher degree of processing capability is required of the embedded controller to compress and decompress the data.

The CCITT standard is primarily focused on black-and-white document transmission and is not ideal for every application. There are denser compression methods, particularly for handling color and gray-scale images. Because of the 34010's programmability, it can be adapted to handle many of these unique compression methods.

Page printers. Laser and other printing technology, such as thermal dye transfer (used in many color printers), is an obvious application for an embedded microprocessor with special graphics capabilities. Figure 8 shows a 34010-based laser printer system. In addition to generating the print image, the processor can perform other functions such as controlling the print engine and the page feeder.

Page printers require the ability to move and manipulate (to the bit level) large bitmaps of data, calculate outline fonts, and emulate dot matrix printers. Many printers also support page description languages such as Postscript or compatible products. For economy the printers typically have used 16-bit microprocessors, but these processors can often be the limiting factor in print speed, particularly when a page description language is used. Even for black-and-white laser printers, the amount of memory required is large due to the relatively high-resolution (typically over 2500 by 3300 dots or one megabit) images they generate. This has led to the almost exclusive use of dynamic memory for the image buffer.

For 6- to 10-page-per-minute laser printers, the 34010 works alone. The on-chip DRAM controller reduces the external logic requirement, and the host interface is used as a general communication port. For performance of 15 to 60 ppm with page description requirements, multiprocessor configurations can be applied to improve throughput. The embedded system's master processor can be either a 16- or 32-bit microprocessor. The master processor handles the page description language interpretation and then passes the graphics processing to the slave processors. The graphics processing performance of the 34010 and the resulting low system cost makes this multiprocessor configuration feasible.

Additional embedded application areas for the 34010 include laboratory and factory data acquisition, optical character recognition, dashboard and cockpit instrumentation, cardiac monitors, radar control systems, process control, robotics, and medical imaging systems.

The 34010 is the first of a family of 32-bit embedded processors blending performance and system integration to support the growth of advanced applications. New family members currently in design will extend the design philosophy of this processor and improve the performance of the next generation of embedded systems.

References

1. R. Noyce and M. Hoff, "A History of Microprocessor Development at Intel," *IEEE Micro,* Feb. 1981, pp. 8-21.
2. C.R. Killebrew Jr., "The TMS34010 Graphics System Processor," *Byte,* Dec. 1986, pp. 193-204.
3. M. Asal et al., "The Texas Instruments 34010 Graphics System Processor," *IEEE Computer Graphics and Applications,* Oct. 1986, pp. 24-39.
4. *TMS34010 User's Guide,* Graphics Products, Texas Instruments Inc., Jan. 1987.
5. R. Pinkham, M. Novak, and K. Guttag, "Video RAM Excels at Fast Graphics," *Electronic Design,* Aug. 18, 1983, pp. 161-168.
6. M. Johnson, "System Considerations in the Design of the Am29000," *IEEE Micro,* Aug. 1987, pp. 28-41.
7. E. Stritter and T. Gunter, "A Microprocessor Architecture for a Changing World: The Motorola 68000," *Computer,* Feb. 1979, pp. 43-52.
8. R.P. Colwell et al., "Computers, Complexity, and Controversy," *Computer,* Sept. 1985, pp. 8-19.
9. R.P. Blake, "Exploring a Stack Architecture," *Computer,* May 1977, pp. 30-38.
10. D. Fairclough, "A Unique Microprocessor Instruction Set," *IEEE Micro,* May 1982, pp. 8-18.
11. D.A. Patterson and C.H. Sequin, "A VLSI RISC," *Computer,* Sept. 1982, pp. 8-20.
12. *TMS34010 3rd Party Guide,* 2nd ed., Texas Instruments Inc., Oct. 1987.
13. *C Executive User's Manual,* JMI Software Consultants, 1986.
14. "Standardization of Group 3 Facsimile Apparatus for Document Transmission," Recommendation T.4, CCITT, 1984.
15. "Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Apparatus," recommendation T.6, CCITT, 1984.

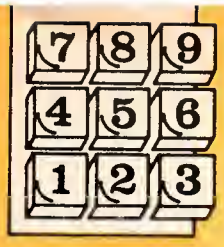
Karl M. Guttag works in the Microprocessor and Microcontroller Division of Texas Instruments. Since 1982 he has been responsible for graphics product definition, including graphics processor architecture and the multiport video RAM. Previously, he was the IC architect of two 16-bit microprocessors and a design engineer of the TMS9918 video display processor. Guttag received his BSEE from Bradley University and his MSEE from the University of Michigan. He was elected a Texas Instruments fellow in 1988. He holds 25 patents in microprocessors and computer graphics hardware and is a member of the IEEE and the ACM.

Thomas M. Albers is responsible for software support of the 340 family of processors. He joined TI in 1980 and has worked with 4-, 8-, and 32-bit microprocessors. He has contributed to the microcode tools, software simulator, and operating environment for the TMS34010. Currently, he is working on strategic software development and TI's next generation of graphics devices. Albers received a BS in electrical engineering from Texas A&M University. He is a member of Tau Beta Pi and the ACM.

Michael D. Asal is a systems engineer in the Microprocessor and Microcomputer Products Division of Texas Instruments. He is currently working on the design of TI's next-generation graphics processor in Bedford, England. Since joining TI in 1982, he has been involved with the specification, architectural development, and design of the TMS34010 and higher performance follow-on devices. He was elected a member of the Semiconductor Group technical staff in 1987. Asal received the BSEE and MSEE from Bradley University.

Kevin G. Rose is a strategic marketing engineer with Texas Instruments. He provides application and marketing support to laser printer, terminal, and facsimile designers. He also manages business development for these markets. Rose received his BS from Brigham Young University and his MBA from the University of Utah.

Questions about this article can be addressed to Tom Albers, Texas Instruments Inc., PO Box 1443, M/S 712, Houston, TX 77001.



FEATURE

The MIPS R3010 Floating-Point Coprocessor

A RISC architecture and VLSI techniques speed operations in a coprocessor closely coupled to its CPU. The 75,000-transistor, hardwired chip executes four instructions in parallel.

Chris Rowen
Mark Johnson
Paul Ries

MIPS Computer Systems

Computer systems for engineering and scientific applications require both fast integer and fast floating-point arithmetic. Unfortunately, while recent 32-bit, very large scale integration (VLSI) processors have set a new cost-performance standard for integer operations,[1] floating-point operations remain either slow or expensive.

At MIPS Computer Systems we solved this problem with the R3010 floating-point accelerator chip. Based on advanced reduced instruction set computer (RISC) architecture and VLSI design techniques, the R3010 packs high floating-point performance into one custom device. It serves as a coprocessor for the R3000 RISC VLSI CPU. It transparently adds floating-point extensions to the CPU's instruction set, by cointerpreting the common instruction stream. These floating-point extensions include all of the same forms as the R3000's integer operations: loads and stores, register moves, compares, arithmetic operations, and branches.

The most dramatic benefit of a closely coupled coprocessor occurs in the large class of scientific and engineering problems that resist code vectorization. These unstructured problems require low latency of both memory references and arithmetic operations. The R3000 architecture sharply reduces effective memory latency by its use of a large, high-speed data cache and block-refill techniques.

The R3010 achieves high-speed arithmetic operations through the interaction of four important design choices:

- Tight coupling of the floating-point unit (FPU) with the CPU reduces the instruction-issuing overhead and helps in the precise handling of exceptional numerical conditions. This choice ensures IEEE compatibility without performance degradation.
- Global optimizing compilers exploit a large set of floating-point registers (sixteen 64-bit registers), eliminating unnecessary computations and memory traffic.
- An implementation not written in microcode (hardwired) boosts the speed of essential operations in IEEE single- (32-bit) and double-precision (64-bit) operations. Consequently, library software written in assembly language handles infrequent operations, such as trigonometric and hyperbolic functions, providing the maximum silicon area for the highest speed implementation of the most frequent operations.
- Independent functional units allow up to four floating-point instructions to execute in parallel.

072-1732/88/0600-0053$01.00 © 1988 IEEE

Available floating-point performance

The execution time of any program can be expressed as the product of the number of instructions executed, the average number of cycles per instruction, and the cycle time of the processor. In designing the R3000/R3010 pair of RISC processor chips, we chose to reduce both cycle time and cycles per instruction. As a result, hardware directly implements the essential floating-point operations of addition, subtraction, multiplication, division, comparison, conversion between formats, absolute value, and negation. System software supplies the more complex functions and while doing so benefits from the fast execution of the underlying arithmetic. Table 1 lists the number of cycles for important floating-point operations in both single- and double-precision formats.

Other single-chip floating-point coprocessors implement these basic operations with lengthy microcode sequences that take three to 13 times as many cycles as the R3010. Table 2 compares the number of cycles required to perform double-precision arithmetic operations on an R3010 and a Motorola MC68882 FPU.[2] We list register-to-register and memory-to-register operations. Note that even the sine function, an assembly language library function on the R3010 but microcoded in the MC68882, runs more that 2.8 times faster on the R3010 at the same 25-MHz clock rate. The cycle counts for both the R3010 and the MC68882 assume zero wait states or cache misses. The R3010 cycle counts include the total latency seen by the program, but the MC68882 cycle counts include only the execution unit times and not instruction fetch overhead. Note also that the MC68882 loads double-precision memory operands into an 80-bit internal representation used for all computations.

Cycle counts show the raw speed of the floating-point hardware. However, they don't necessarily predict the effectiveness of the FPU as part of a computer system running real benchmarks. Figure 1 and Table 3 compare a system based on the R3000 and R3010 with three other systems that are popular in the engineering and scientific community. These are the DEC VAX 8700 (under the VMS operating system), the MC68020-based Sun 3/260 with an MC68881 (Unix BSD 4.2), and the Sun 4/260 RISC processor with a custom floating-point controller for the Weitek 1164/1165 floating-point chip set (Unix BSD 4.2). The R3000/R3010 system runs at 25 MHz with 64-kilobyte instruction and data caches and a block-mode memory (Unix System V). The Sun 3/260 MC68020 runs at 25 MHz, the MC68881 at 20 MHZ, and the Sun 4/260 at 16.67 MHz.

We tabulated the execution times for four widely known benchmarks: double-precision Linpack (Fortran), double-precision Whetstone (Fortran), the popular SPICE-2G.6 circuit simulator on the Mosamp2 test case (Fortran), and DoDuc's large nuclear reactor simulator (Fortran).[3] We normalized the performance figures to the speed of a Digital Equipment Corporation VAX 11/780 running the benchmarks under the VMS operating system. (We chose VMS for the normalizing factor rather than Unix because the Unix/Fortran compiler for the VAX, the f77, is known to be exceptionally inefficient).

Table 1. Cycle count for R3010 operations. (A cycle takes 40 ns.)

Operation	Single	Double
Add	2	2
Multiply	4	5
Divide	12	19
Move, absolute value, negate	1	1
Compare	2	2

Table 2. R3010 versus MC68882 cycle counts.

Operation (double precision)	Reg—reg op reg		Reg—reg op mem	
	R3010	MC68882	R3010	MC68882
Add	2	26	4	36
Multiply	5	46	7	52
Divide	19	78	21	78
Sine	111	320	113	320

Coprocessor architecture

The RISC architecture includes tightly coupled coprocessors as integral to its design. Each of the four coprocessors (called 0 through 3) simultaneously interprets the common stream of instructions fetched by the R3000 CPU, performs operations on private register files, and loads and stores operands/results in the common data cache. Coprocessor 0, which is implemented on the R3000 processor chip itself, handles memory management and system control functions. Coprocessor 1, the R3010 chip, is dedicated to binary floating-point operations. Coprocessors 2 and 3 remain undefined and available for future architectural extensions. The shared execution by the coprocessors and the CPU reduces the complexity and pin count of the coprocessors without sacrificing speed.

For Coprocessor 1 (R3010) loads, the CPU adds the base register and offset to form a virtual address, translates it to a physical address (via Coprocessor 0), and sends the physical address to the data cache. The CPU checks the cache tag while the R3010 captures the data. The CPU manages refills from main memory on cache misses. If desired, operands can also be moved between the CPU's general-purpose registers and the R3010's operand or control registers.

Floating-point instructions usually specify a destination register and two source registers, along with a format. The R3010's registers are referenced by load, store, and processor move instructions as thirty-two 32-bit values, but arithmetic operations treat the registers as sixteen 64-bit values. Each of the 16 registers holds either a single- (32-bit) or double-precision (64-bit) operand. Floating-point control registers include a read-only register for revision identity and a status register containing rounding, IEEE exception control, and condition bits. Figure 2 summarizes the register organization and the three floating-point instruction types.

The R3010 provides floating-point computation instructions for single- and double-precision formats:

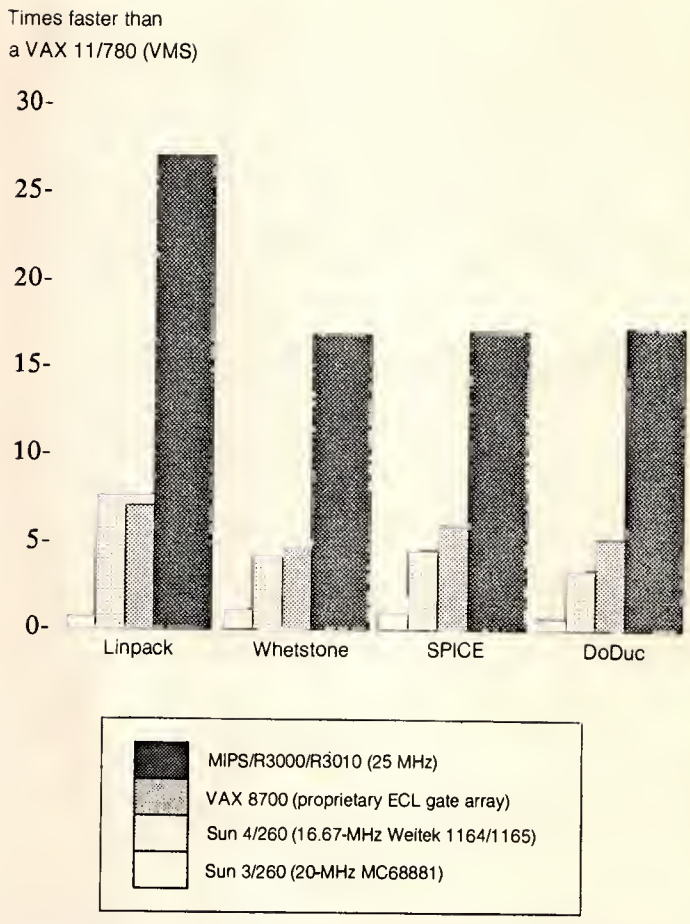


Figure 1. Floating-point system performance comparison.

add, subtract, multiply, divide, compare, absolute value, and negate. The R3010 also converts single-precision, double-precision, and (32-bit) integer formats. The floating-point branch, nominally a part of the FPU instruction set, is actually implemented entirely by the CPU, which uses the condition value gener-

Table 3. System performance comparison for floating-point benchmarks.

System	Linpack (MFLOPS)	Whetstone (KWhets)	SPICE (seconds)	DoDuc (seconds)
MIPS R3010	3.8	14000	3.8	108
VAX 8700	1.0	3950	13	359
Sun 4/260	1.1	3540	17	540
Sun 3/260	0.11	1230	60	2212
VAX 11/780	0.14	830	65	1872

ated in an earlier R3010 compare instruction. The R3010's operations provide the basis for systems conforming with the requirements and recommendations of IEEE arithmetic.[4] In addition, the chip adopts the RISC approach in handling floating-point exceptions. Hardware directly implements all frequently used operations but traps to closely coupled system software to process infrequent, complex IEEE exceptions and rare denormalized values or Not-a-Number values.

Organization and implementation

Four independent arithmetic functional units in the R3010 (register, add, divide, and multiply) interact with a scheduling and managing *control unit*. The control unit watches the transfers between the R3000 CPU and the instruction cache and interprets each floating-point operation. Whenever the CPU requires the result of a floating-point instruction before the operation is complete, the control unit signals the CPU to wait. If the CPU detects an exceptional condition (such as a cache miss or an external interrupt), it shuts down the floating-point execution pipeline so that unfinished instructions can be restarted later without numerical inconsistencies. The control unit also schedules the execution of each instruction by the four arithmetic units.

- The *register unit*'s register file can perform two 64-bit operand reads, one 64-bit write result, and one memory-load data write in one cycle. Although, logically, four separate ports exist, a two-port design that is accessed twice per cycle actually implements the register file.
- The *add unit* executes add, subtract, convert, compare, negate, and absolute value instructions. It also performs the final IEEE rounding step of multiply and divide operations. This unit includes the exponent data path, which computes the 8 bits (single precision) or 11

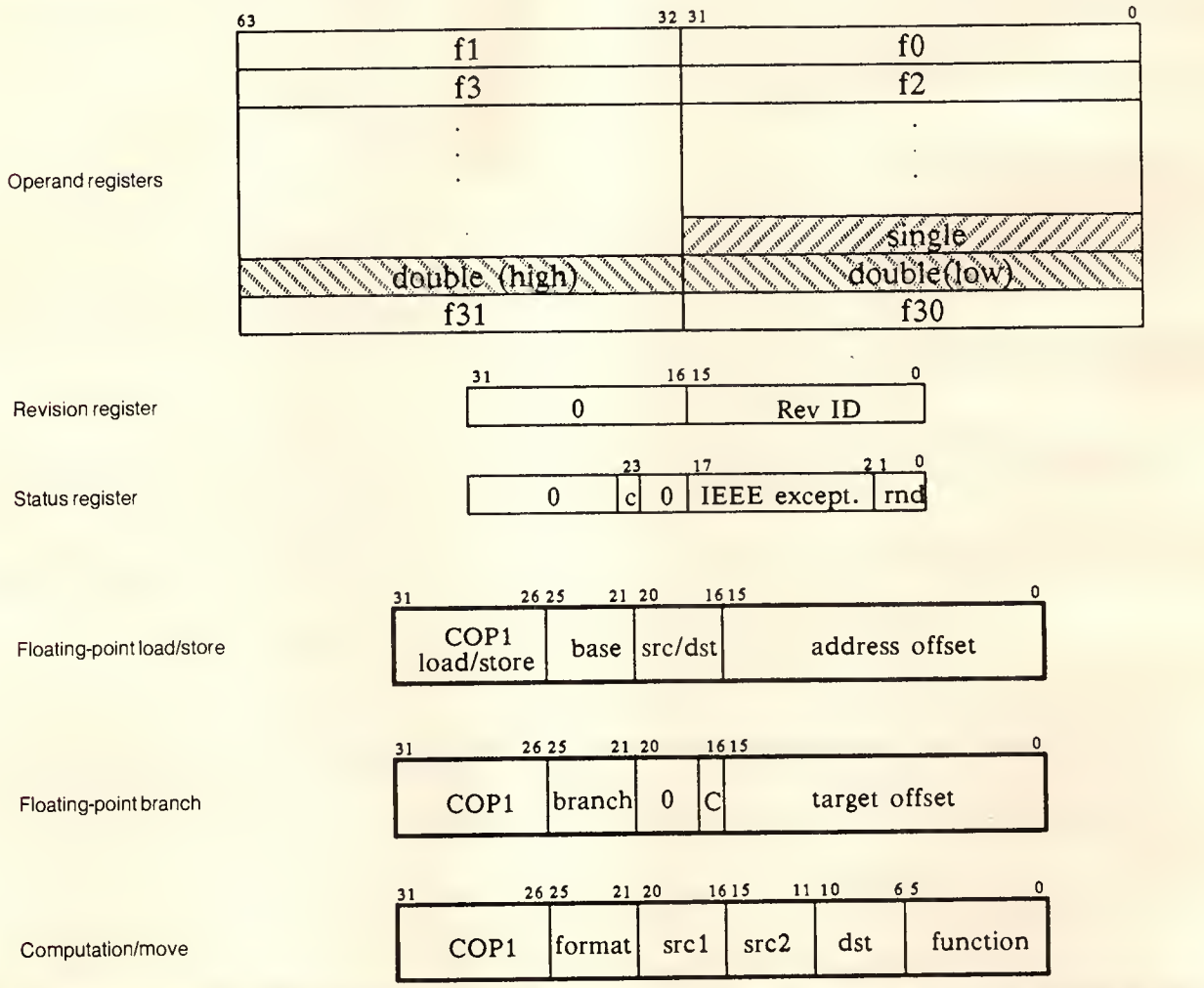


Figure 2. R3010 register and instruction formats.

Implementation of the Multiply Unit

The R3010 uses a slightly enhanced version of the traditional shift-and-add algorithm for multiplication. It computes an accumulated sum of partial products, where the *n*th partial product is the multiplicand shifted *n* bits right and logically ANDed with (times) the *n*th most significant bit of the multiplier. This simple hardwired version of the pencil-and-paper multiplication algorithm for binary numbers increases in speed with three modifications:

1) Booth-encoded[5,6] multiplier bits reduce the number of shift-and-add iterations from 56 to 28, cutting the multiplication time in half.

2) Carry-save adders speed the iteration because no carry-chain delay is incurred.

3) The hardware computes twice again as many bits per iteration, by using pairs of carry-save adders instead of single adders. One member of a pair processes even-numbered partial products, and the other member processes odd-numbered partial products.

The carry-save adder delay is small enough that inputs propagate through five carry-save adders (five iterations of the multiply algorithm take place) in one R3010 cycle. Therefore a double-precision (56-bit) mantissa multiply operation requires $[(56/2)/2] = 14$ iterations, at five iterations per cycle, or 2.8 cycles. Overhead associated with input latching, Booth encoding, sticky-bit computation, control delays, and result drive costs about one cycle more, so the multiply unit needs a little less than four cycles to produce its double-precision result in carry-save form. Another cycle in the add unit produces the final, IEEE rounded result.

Implementation of the Divide Unit

Unlike the multiply unit, the divide unit must perform a full carry-propagate subtraction on each iteration to determine the sign of the partial remainder for the next iteration. Or must it?

A method produced by Atkins,[7] called Higher Radix Redundant SRT Division, exploits an encoding technique in which m symbols represent n values ($m > n$). This method produces a divider that does not need a full carry propagation in each iteration. The R3010 uses a variant of this scheme (a radix-4, redundant, nonrestoring algorithm) employing the five symbols, 2, 1, 0, —1, and —2. The chip uses carry-save adders in the mantissa instead of carry-propagate adders, yielding a significant speedup: Each R3010 cycle sees two iterations performed (4 bits of quotient per cycle).

Maintaining the partial remainders in carry-save form makes the task of selecting the next quotient digit more difficult. The R3010 examines the 9 most significant bits of the partial remainder and the divisor to produce the next quotient digit. The two 9-bit pieces of the partial remainder are reduced from carry-save form into a single binary number in a fast, 9-bit carry-propagate adder. The 9-bit sum transfers with the divisor into a lookup table to find the next quotient digit. The mantissa carry-save adders and the quotient determining logic (9-bit adder + lookup table PLA) operate in a parallel, pipelined manner, similar to a board-sized ECL (emitter-coupled logic) divider described by Taylor.[8]

At the end of a divide, the full mantissa's partial remainder (in carry-save form) moves to the add unit for carry-propagate addition. If the remainder is nonzero, the R3010 makes guard, round, and sticky adjustments to the quotient and sets IEEE status flags (Inexact and so on). The redundant-encoded quotient then transfers to the adder for final carry propagation and IEEE rounding. A double-precision divide requires a total of 19 cycles (12 cycles for a single-precision divide).

bits (double precision) of exponents for all arithmetic operations. The exponent data path used in the first cycle of arithmetic operations computes the approximate exponent and mantissa alignment; in the final cycle it computes the final exponent result achieved after rounding.

• The *multiply unit* computes the product of the mantissa portion of its operands. The multiplier retains the most significant 56 bits of the 106-bit product of two 53-bit operands in less than four cycles, or about 14 bits per cycle. (Refer to the accompanying box for more information on the multiply and divide units.)

For special cases, the control unit adjusts the issue schedule to maintain the illusion of in-order instruction completion.

• The *divide unit* uses a radix-4, SRT-division algorithm to produce two quotient bits per halfcycle or four quotient bits per cycle. This method, proposed by Atkins[7] and enhanced by Taylor,[8] uses a redundant encoding of the quotient as a sum of digits with values 2, 1, 0, —1, and —2.

Results of both the multiplier and divider appear in carry-save form. They are returned to the adder over the two operand buses for final carry propagation and rounding. A separate path exists for the guard, round, and sticky bits (GRS) required for IEEE rounding. Figure 3 shows the interaction of the five units, including the width of the major data buses.

The autonomy of the add, multiply, register, and divide units allows them to run in parallel. Concurrent instructions generally do not conflict for resources until the rounding stage. Based on the latency of each operation, the control unit schedules instructions to ensure that no two will need the exponent data path or the rounding function of the add unit at the same time. The floating-point architecture requires that instructions must appear to complete in the order they were issued. However, the control unit recognizes the special cases of operations with exceptional results, conflicts for one functional unit, and data dependencies between operations. In these instances it adjusts the issue schedule to maintain the illusion of in-order instruction completion.

Figure 4 shows the overlapping of add, multiply, and divide (all in double precision) operations. The R3010's register unit can also execute memory operations while arithmetic units are busy, as shown by the load and store operations intermixed with the arithmetic. If the add operation had needed the result of the multiply operation as a source operand, the control unit would

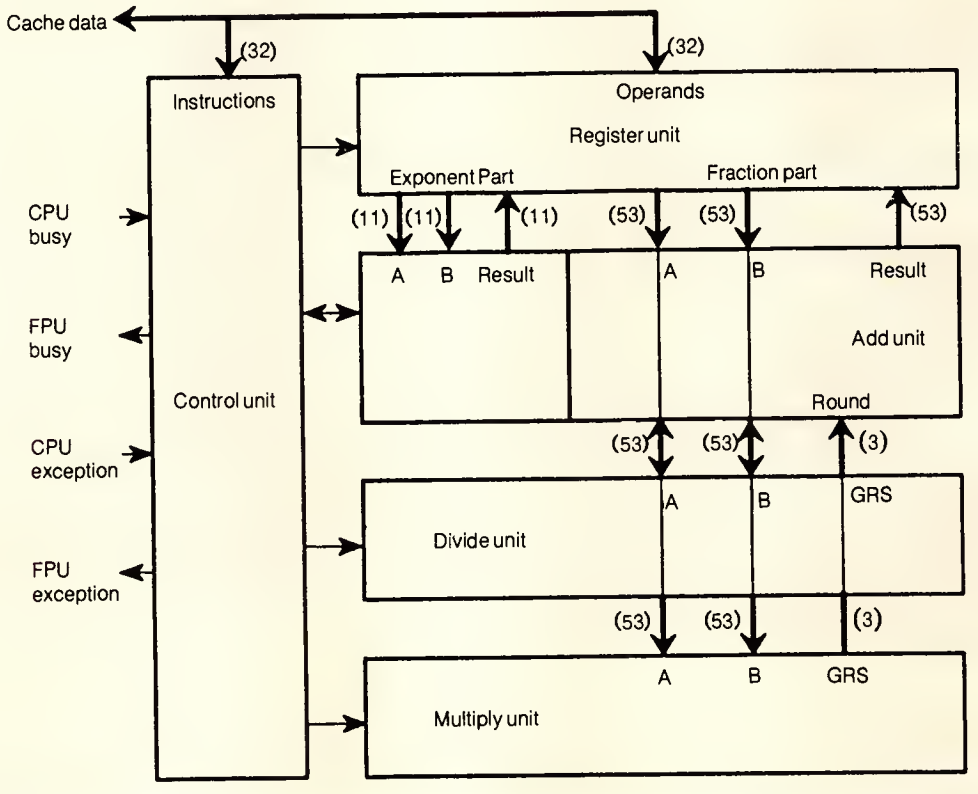


Figure 3. R3010 block diagram.

have signaled the CPU to stall until the multiplication was complete. Multiple instruction overlapping benefits only those scalar floating-point programs with much data dependence between operations. Tightly coded floating-point routines can often exploit this parallelism. The MIPS Fortran, C, Pascal, and Ada compilers attempt to maximize parallelism by doing so.

The second-generation R3010 FPU and the R3000 CPU devices derived from the original R2010 and R2000 CMOS (complementary metal oxide semiconductor) chips. The R3000-generation chips include enhanced cache and main memory interfaces and are fabricated in a higher performance, double-metal CMOS technology. The design reduced average lithographic features to 1.6 μm wide; transistor lengths to below 1 μm; and the gate oxide thickness to about 225 angstroms.

To achieve the aggressive cycle speed of 40 nanoseconds (25 MHz), we chose a labor-intensive, hand-optimized design methodology. This approach allowed us to use high-performance, unorthodox circuits and physical layouts that are unavailable in standard cells, gate arrays, or silicon compilers. As a result, the cost in manpower and design time increased to over 25 man-years and 16 months from design start to first customer shipment. Massive circuit and logic simulation, plus electrical and layout rule checking, produced fully functional parts, over the full temperature and voltage ranges, on the first silicon. Within 10 days of first silicon, we installed floating-point chips into working systems and then ran SPICE simulations of further "tweaks" to the circuit design.

A labor-intensive, hand-optimized design methodology allowed us to use high-performance, unorthodox circuits and nonstandard physical layouts.

Figure 5 is a die photograph of the R3010. The register, add, divide, and multiply units align vertically on the right side of the chip, forming one 53-bit mantissa data path that extends the entire height of the device. A separate 11-bit data path in the add unit processes exponents. Second-level-metal power buses partition the mantissa path into three 18-bit sections. These power buses are vital for reducing transient supply voltage drops across the mantissa path. An example would be when the A bus and B bus (a total of 106 lines) switch simultaneously. The transient C * dV/dt supply current for switching these lines is about 310 milliamperes, sufficient to induce significant IR voltage drops unless abundant orthogonal metal power busing is provided. The large power buses also improve electromigration reliability.

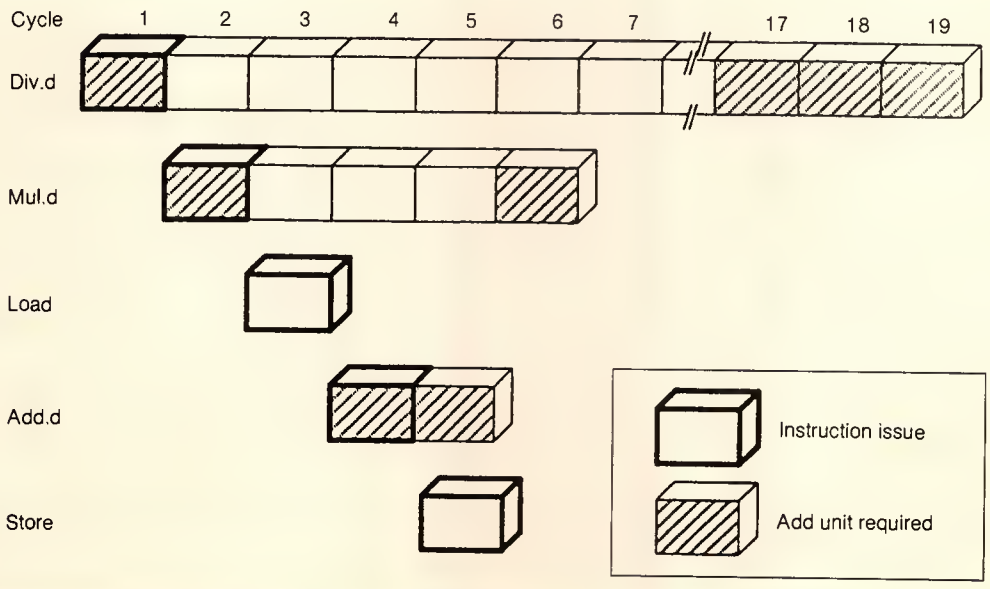


Figure 4. Overlapping arithmetic and memory operations.

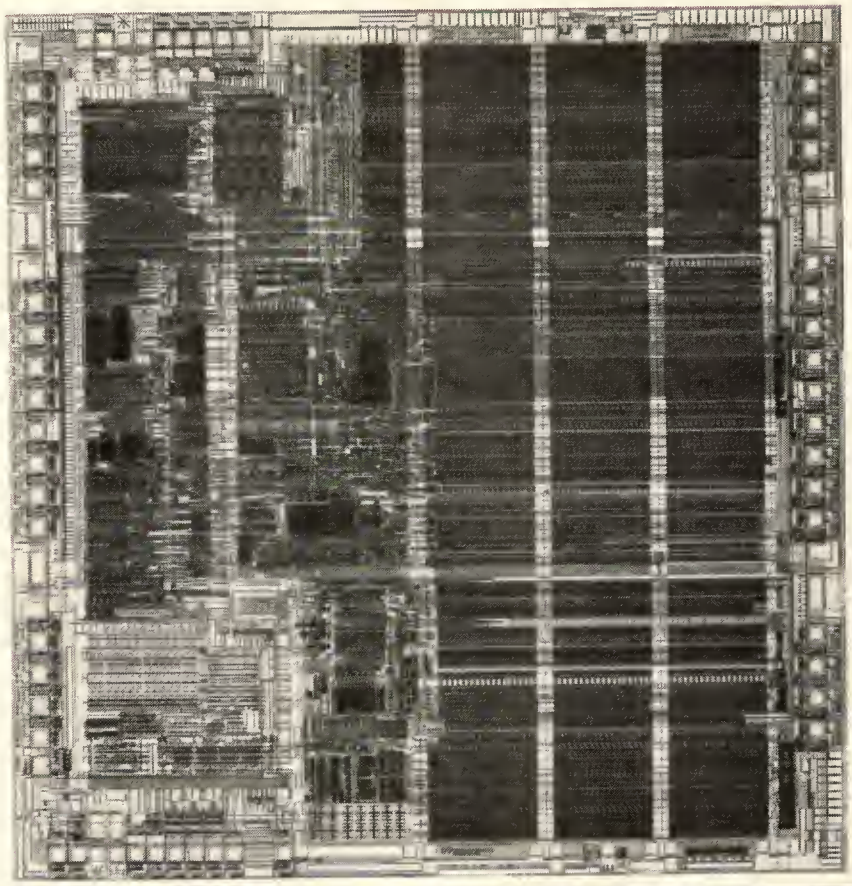


Figure 5. Photomicrograph of the R3010.

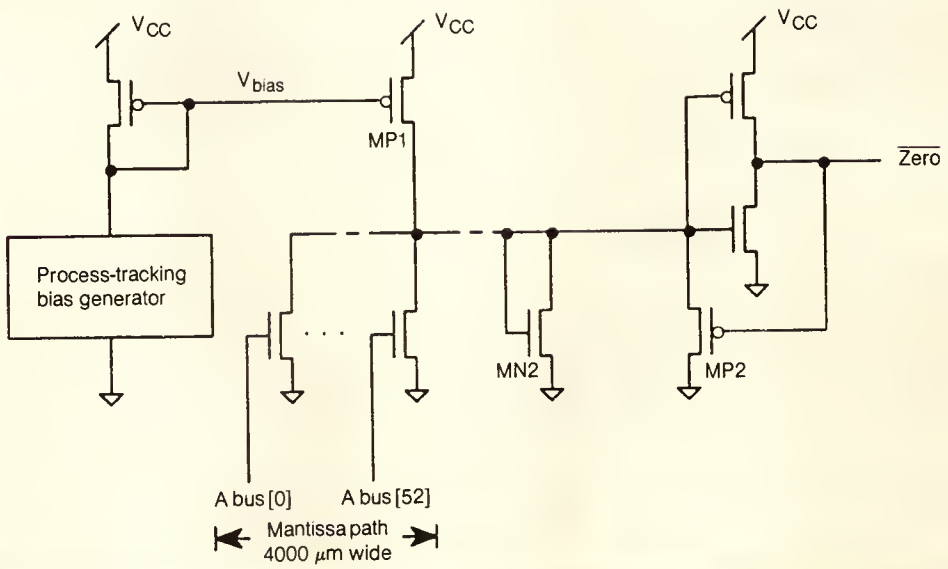


Figure 6. Mantissa zero-detector circuit.

Figure 6 depicts one of the unconventional circuits used in the R3010, the mantissa zero-detector. Because its inputs can become valid at several different points within the cycle, and its outputs are required a fixed combinational delay later, we did not feel the more common precharge/discharge implementation was appropriate. (Generating all of the required precharge timings, and selecting among them, proved to be too slow and complex.) We used a pseudodepletion static NOR gate instead. The PMOS (*p*-channel MOS) pullup MP1 and the bias generator provide a constant current load similar to depletion pullups in NMOS (*n*-channel MOS) designs. The diode-connected transistor MN2 and negative-feedback device MP2 clamp the voltage swing on the high-capacitance, distributed NOR node, trading noise margin for increased speed. The bias network adjusts the PMOS pullup current to compensate for variations in temperature, supply voltage, and transistor processing parameters (VT, Tox, Leff) to maintain the maximum possible bias current (hence the highest speed), which still produces an acceptable logic zero. We used this pseudodepletion style several times on the chip. It appears in the programmable logic arrays, the exponent adder, the divider's quotient logic, and other locations where the chip requires fast, large fan-in gates.

Figure 7 shows the input latch circuit used in the R3010. A dynamic, differential sense amplifier compares the bond-pad voltage against a reference voltage equal to 0.3 V_{CC} (1.5 volts). The more conventional circuit, a ratioed inverter followed by a transparent latch, suffers from long setup and hold times, process-sensitive trip thresholds, static power consumption, and longer propagation delay. These problems have traditionally plagued MOS circuit designs that interface to bipolar TTL (transistor-transistor) logic levels. In the R3010 circuit the bond-pad voltage feeds directly into a sample-and-hold pass transistor, giving quite small setup and hold times. Typically, 1.0-ns setup and 1.0-ns hold times suffice for correct operation. The circuit's input trip level is insensitive to transistor parameters and temperature, giving a stable external interface over a wide range of process and environmental conditions. Differential, positive-feedback, dynamic latching produces full-rail CMOS outputs without consuming any DC power. This is especially advantageous when large numbers of TTL input latches are needed, since we avoid the traditional speed-versus-power trade-off. Finally, the high-gain bandwidth attained by the positive feedback topology produces faster TTL-to-CMOS conversion than is possible with conventional nonfeedback designs.

The R3010 floating-point coprocessor, packaged in an 84-pin chip carrier, typically consumes 200 mA of power. Worst-case (5.5V, 0 degrees Centigrade, all outputs loaded and switching) power dissipa-

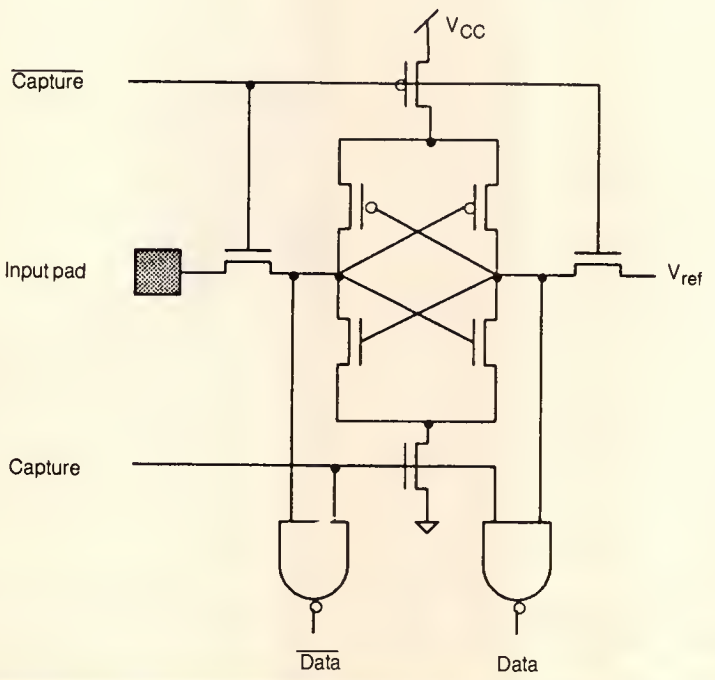


Figure 7. Clocked input buffer circuit.

tion can reach 3.8 watts if the device received narrow, polysilicon (short-channel) processing at this specified minimum. Curiously, the R3010 contains *only* 75,000 transistors on a rather large die (0.331 by 0.347 inches including scribe streets). This size results mainly from the lack of large PLAs or microcode ROMs, one benefit of reduced instruction set design. It is also a consequence of the hand-optimized methodology: Each transistor present is tuned for maximum speed and is therefore rather large.

Acknowledgments

Advances in processor design don't come without enormous efforts expended by many people. Ed Hudson, Craig Hansen, and Tom Riordan defined the R3010. Craig specified the architecture, while Tom, Paul Ries, and John Kinsel came up with the numerical algorithms and logic design. Ray Kunita managed the physical implementation. Tom Vo and Mark Johnson developed the circuit implementation. Mike Wageman, Claudia Kyle, Orlando Hernando, Ben Leone, and Ron Genenbacher drew the layout, while Roger March and Dave Van't Hof developed and ran the tools to verify the design. Chris Rowen wrote the design diagnostics and tested conformance with the IEEE 754 floating-point standard. Keith Garrett and John Cawley developed test hardware and software. Earl Killian and Kevin Enderby provided valuable advice on floating-point performance and IEEE software. We especially thank Tom Vo for comments on this article.

References

1. C. Rowen et al., "RISC VLSI Design for System-Level Performance," *VLSI Systems Design,* Mar. 1986, pp. 81-88.
2. *MC68882, HCMOS Enhanced Floating-Point Coprocessor Technical Summary,* Motorola, Inc., 1987.
3. N. DoDuc, FORTRAN Central Processor Time Benchmark, Version 13 (electronically distributed), Framentec, June 1986.
4. *ANSI/IEEE Std. 754-1985, Standard for Binary Floating-Point Arithmetic,* IEEE, New York, Aug. 12, 1985.
5. A.D. Booth, "A Signed Binary Multiplication Technique," *Qt. J. Mech. Appl. Math.,* Part 2, 1951.
6. S. Waser and M.J. Flynn, *Introduction to Arithmetic for Digital Systems Designers,* Holt, Rinehart, and Winston, New York, 1982, pp. 133-139.
7. D. Atkins, "Higher-Radix Division Using Estimates of the Divisor and Partial Remainders," *IEEE Trans. Computers,* 1968, pp. 925-934.
8. G. Taylor, "Radix 16 SRT Dividers With Overlapped Quotient Selection Stages," *Proc. Seventh IEEE Symp. Computer Arithmetic,* June 1985, pp. 64-71.

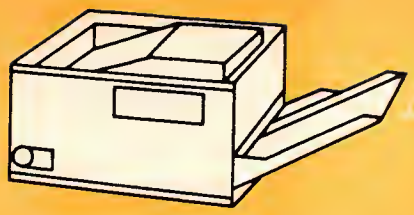
Chris Rowen works with the design and validation of VLSI-based systems at MIPS Computer Systems. For several years previously he worked on static RAMs and microprocessor technology at Intel Corporation. He received his AB in physics from Harvard University and his MS and PhD from Stanford University.

Mark Johnson currently defines and designs advanced microprocessor circuits and systems at MIPS. He has worked on the R2010/R3010 floating-point coprocessor design, assuming responsiblity for the multiply unit, control unit, clocks, phase-lock loop, and I/O circuits. Prior to this, he worked in the memory design group at Mostek Corporation, where he designed the MK41H68 family of 20-ns static RAMs and codesigned a 1M-bit, 25-ns, static-column, CMOS dynamic RAM.

Johnson received the BSEE degree from Rice University and the SM degree from the Massachusetts Institute of Technology.

Paul Ries joined MIPS to work with logic design and verification of the R2010 and R3010 floating-point coprocessors and the R3000 CPU. Currently, he defines and designs advanced microprocessor products. Earlier, he worked for Intel Corporation on signal-processing microprocessors, CAD, and the 80386 microprocessor projects. Ries received his BSEE from MIT.

Questions about this article can be directed to Chris Rowen, MIPS Computer Systems, 930 Arques Avenue, Sunnyvale, CA 94086.



FEATURE

Intel's 80960: An Architecture Optimized for Embedded Control

Intel's internally developed 80960 architecture allows embedded system designers to take advantage of silicon technology advancements without architectural limits. The 80960, developed from scratch for embedded control applications, eliminates architectural obstacles to state-of-the-art implementation techniques that allow parallel and out-of-order instruction execution.

In introducing the 80960 architecture, I distinguish between the architecture and the microarchitecture of an implementation. A microarchitecture is an actual implementation of the architecture's instruction set and programming resources. Different microarchitectures may have different pipeline construction, internal bus widths, register set porting, cache parameterization (two-way, four-way, and so on), and degrees of parallelism, or may not execute instructions out of order. The architecture is specified in such a way that wide latitude is available for future microarchitectural advancements. In this way both very high performance and highly integrated controllers can be built using the 80960 architecture.

Principally, the 80960 architecture allows, for at least the next decade, silicon technology and microarchitectural advances to translate directly into increased performance without architectural limitations. While the common performance target of typical RISC architectures is an execution speed of one instruction per processor clock cycle, the 80960 architecture targets the execution of multiple instructions per clock cycle (fractional clock cycles per instruction). By defining an architecture that supports parallel instruction execution and out-of-order instruction execution, we do not constrain performance advances to the system clock cycle.

Additionally, the 80960 has been optimized for the wide range of applications that are unencumbered by a need to be compatible with an existing embedded control architecture. These applications are often very cost sensitive, require a different mix of instructions than reprogrammable applications, have demanding interrupt response requirements, and use real-time executives rather than full-blown operating systems. With these factors in mind, we developed the 80960 while avoiding architectural constructs that would restrict an implementation's capability to execute multiple instructions in one clock cycle.

Executing instructions in one clock cycle is not fast enough for this standard-core architecture. Its parallelism and out-of-order execution promise fractional clock rates in future implementations.

David P. Ryan
Intel Corporation

072-1732/88/0600-0063$01.00 © 1988 IEEE

The architecture also allows application-specific extensions such as:

- instruction set extensions (floating-point operations),
- special registers,
- larger caches,
- multiple caches,
- on-chip program and data memory,
- a memory management and protection unit,
- fault-tolerance support,
- multiprocessing support, and
- real-time peripherals (DMA, analog/digital, serial ports).

The 80960's instruction set has also been optimized for embedded control applications. It offers Boolean operations more powerful than those of the 8051 family. Single instructions access frequently executed functions for increased code density and performance. They include CALL, RET, Compare_and_Branch, Conditional_Compare, Compare_and_Increment or Decrement, and Bit Field Extract.

A priority interrupt structure simplifies the management of real-time events, and, along with a user/supervisor model, supports fast, safe, real-time kernel operation. A generalized fault-handling mechanism simplifies the task of detecting errant arithmetic calculations or other conditions that typically require a significant amount of user code runtime support. The 80960 does not require sophisticated, optimizing high-level-language compilers to achieve high performance. However, no obstacles to performance enhancements via a good optimizing compiler exist.

Since products based on the 80960 have high performance levels, even without code optimization, many users will attain their required system performance with 80960 products without having to understand the parallelism of the implementation they are using.

Architecture overview

The 80960 can best be described as a register-rich, load/store architecture with an instruction set designed to let implementations exploit pipelining and parallel execution strategies. Direct embedded control support includes:

- an optimized instruction set,
- a flexible interrupt structure,
- a user-supervisor model,
- a powerful fault-handling structure, and
- multiprocessing hooks and debug support.

The architecture extends easily.

Figure 1 shows a logical block diagram of the architecture's programming resources. The 32-bit memory space is flat. Data moves between memory and registers via load and store instructions. Processing elements surrounding the registers manipulate parallel data; they receive their instructions from the

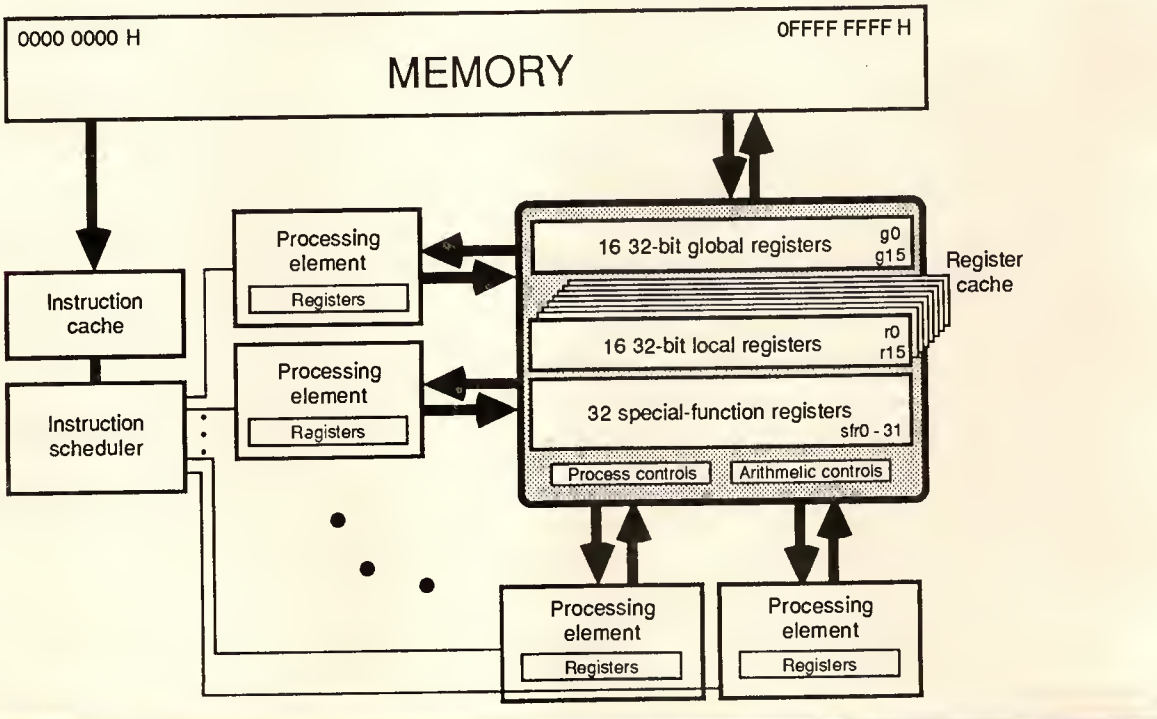


Figure 1. Block diagram of the 80960 architecture.

instruction sequencer.

The instruction sequencer reads multiple instructions simultaneously from an instruction cache and presents the instructions in parallel to the processing elements as appropriate. When the instruction stream or an asynchronous event requires a context switch, the local variables from the suspended procedure move to the register cache. Memory accesses to save previously suspended register sets occur only when the register cache is filled. The implementation determines the number of architecturally transparent register sets that can be cached.

We expect different implementations of the processing elements in an 80960-based controller—optimized for specific applications—will evolve. We defined a standard core architecture to maintain binary-compatible instruction sets across all implementations for leveraging compiler and real-time kernel development. Subsequent references to the 80960, without an alpha suffix, refer to the architecture. References to an 80960*XY* pertain to actual implementations of the core architecture, which may contain architectural extensions and/or on-chip peripherals. (See the accompanying box for a discussion of three implementations.)

Implementations of the 80960

As an example of an actual implementation of the core architecture, consider the first 80960 implementation, the 80960KB controller. The KB provides architectural extensions such as floating-point operations (Figure A). Its on-chip floating-point unit implements the IEEE floating-point standard, including transcendental support. Floating-point performance exceeds 4 million Whetstone operations per second at 20 MHz. The 80960KB integrates a 512-byte instruction cache and an interrupt controller on chip.

Another member of the family, the 80960KA controller, fits the KB socket but lacks the on-chip floating-point unit. The 80960MC controller, a military-qualified version similar to the KB, adds Ada tasking support and a memory management and protection unit.

The 80960KA, KB, and MC microarchitecture sustains execution of up to one instruction per clock cycle at 20 MHz (20 native MIPS). It performs a variety of benchmark programs seven to 10 times faster than a VAX 11/780 (7 to 10 VAX MIPS).

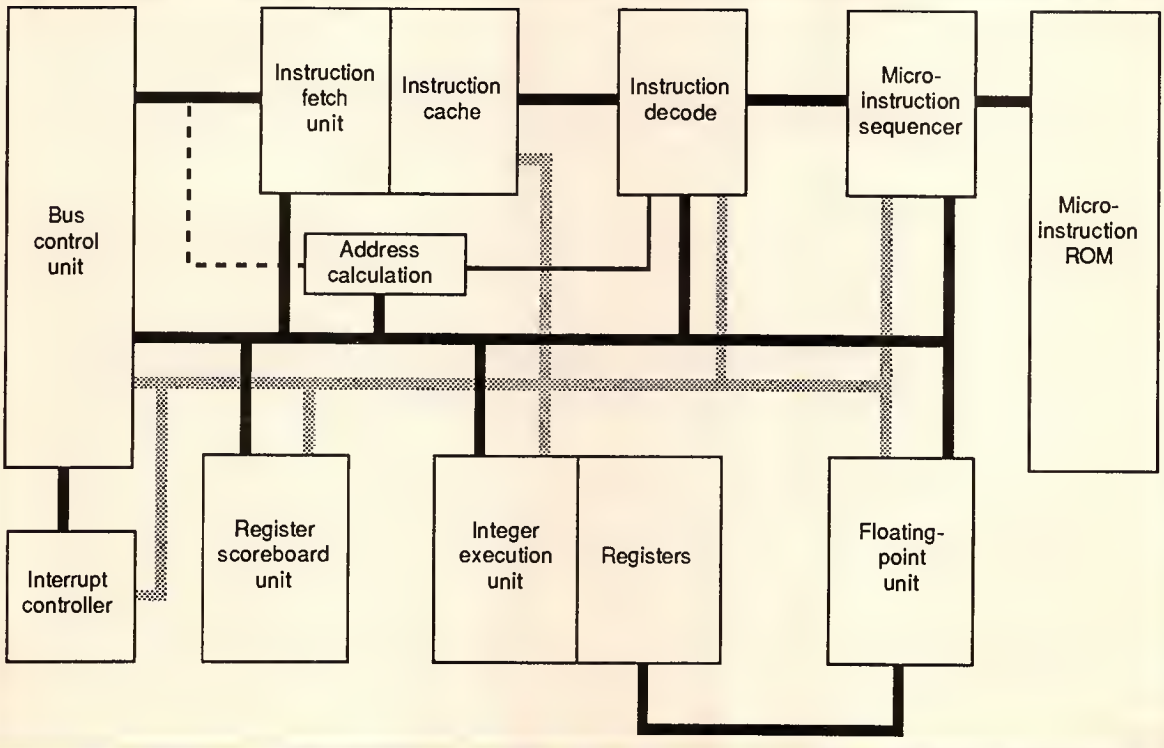


Figure A. Block diagram of the 80960KB controller.

Register model

The user directly accesses thirty-two 32-bit general-purpose registers and 32 special-function registers (SFRs). (Refer to Figure 1.) Of the 32 general registers, 16 are global registers and 16 are local registers. Data in the 16 global registers remain visible and unchanged when crossing procedure boundaries, characteristics exhibited by "normal" registers in other architectures. The CALL instruction caches the local registers and the RET instruction restores them. The SFRs provide a real-time register interface to on-chip peripherals. The SFRs, the contents of which are not defined by the architecture, control implementation-specific hardware.

When a procedure call occurs, data in the local registers automatically move to a register cache. Thus, the called procedure is not required to explicitly save the local registers. When the called procedure executes a return instruction, the data in the local registers prior to the call are restored. The call/return sequence does not affect the global registers, which can be used to pass parameters and results between procedures.

A large, general-purpose register set greatly reduces the number of memory requests required to perform a task. Various optimization techniques can use large-register sets to remove data dependencies that would otherwise prohibit parallel instruction execution. For example, a hypothetical implementation of the architecture could provide parallel execution of two ADD instructions. Having many general-purpose registers with which to work simplifies code optimization so that neither ADD instruction references a source that was the destination of the other ADD instruction. Under such circumstances, two ADD instructions per cycle could be sustained.

Note that such program optimizations are not required. Any program using the architecture's core instruction set and not referencing the SFR address space or external I/O, whether optimized or not, will function correctly on any implementation without modification. Even if the optimization rules are radically different between implementations, code that is optimized for one implementation will run correctly on another implementation without modification or recompilation.

The architecture also guarantees that data dependencies will be correctly handled without programmer intervention. For example, execution of an arithmetic instruction will be delayed if it uses a register that is waiting for data to be loaded from memory. However, other instructions that do not use the register in question could be executed immediately with all data dependencies automatically maintained. This capability is only beneficial when a large register set is present to remove avoidable data dependencies.

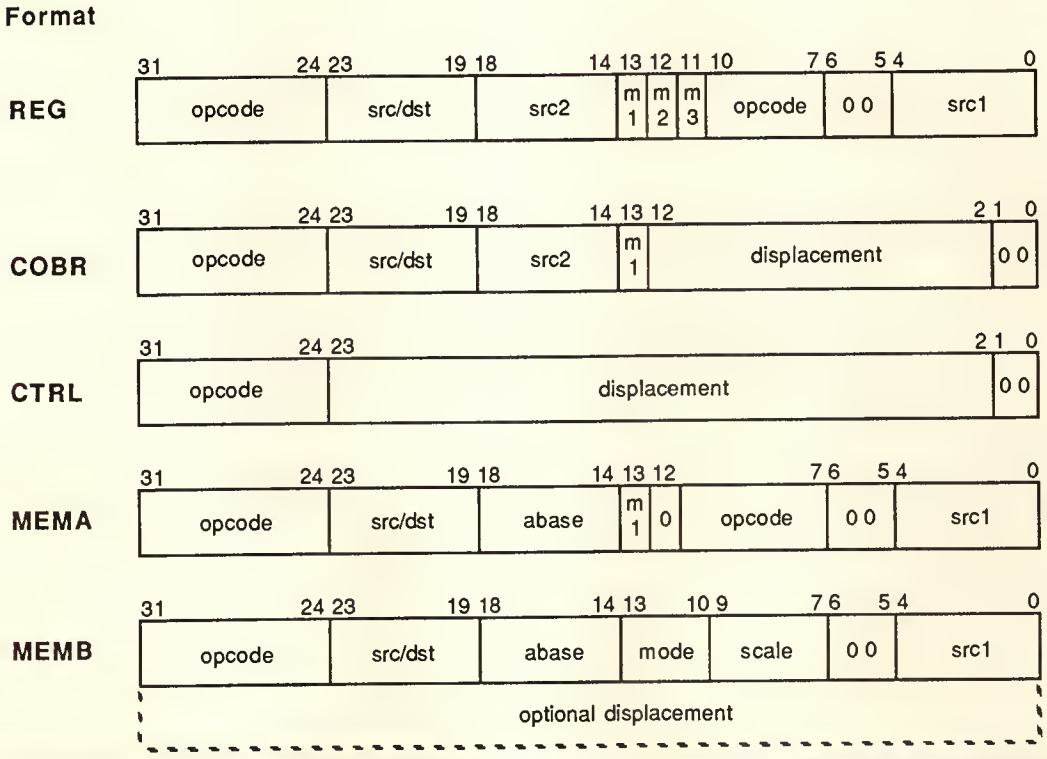


Figure 2. Opcode encodings.

Core instruction set

The 80960 instruction set is similar in design to engines in reduced instruction set computers. Because the 80960 was designed to avoid performance bottlenecks resulting from instruction decoding times, all opcodes are 32 bits (one word) in length and must be aligned on word boundaries. The only two-word (64-bit) instruction format loads 32-bit immediate constants and assists effective address calculations. Generally, load, store, and control instructions access memory. All other instructions access only registers.

Thirty-two-bit opcodes provide tremendous flexibility in instruction encoding. However, performance penalties associated with code size can occur when often-used complex instruction sequences are not available in a one-word format. Large code size not only increases system cost due to larger memory requirements but also results in penalties in cache utilization and bus-bandwidth requirements.

The 80960 includes one-word multifunction instructions to avoid such code density problems that increase memory requirements and to allow complex functions to be parallelized. For example, the CALL and RET instructions provide one-word encodings for sequences that otherwise require several instructions. Implementations of the architecture could perform the CALL/RET operations in parallel with other instruction execution. Or, the processor could execute from an on-chip ROM a similar sequence of simple 80960 instructions that the user would have to write if CALL/RET instructions were not in the architecture. Executing the code from permanent on-chip storage results in higher performance than does requiring the instructions to be fetched and cached every time they are used. In addition to ensuring higher performance, or possibly parallel performance, implementations of such functions as ROMed assembly code sequences—triggered by a one-word opcode—are more silicon efficient than are increases of on-chip cache sizes to deal with poor code density. For example, a ROM cell is typically one fourth the size of a RAM cell.

The availability of complex instructions in the architecture does not prohibit the programmer from bypassing them if simpler functions are desired. For example, a BAL (Branch_and_Link) instruction can be used to call a procedure that does not require a new set of local registers. The BAL instruction saves the next instruction pointer in a register and branches to the specified destination. When the procedure is ready to return, it executes an indirect branch to the return instruction pointer that was saved. It is likely that BAL will always be faster than CALL since no local registers are being saved. The programmer can use whichever method best suits the circumstances.

Figure 2 shows the 80960 instruction encodings. Most instructions appear in the REG format, which specifies an opcode and three registers/literals (one of 32 registers, or a constant value in the range 0 to 31). The COBR format specifies a set of compare and branch instructions. The CRTL format covers branch and call instructions. The MEM formats support load and store instructions.

Much of the instruction set is what one would expect to encounter in all processors (ADD, MUL, SHIFT, BRANCH); however, some instructions deserve special mention.

- The register-register move instructions transfer one, two, three, or four register values. The same is true for the load and store instructions (for example, LDQ loads four words into four registers).
- In addition to the normal shift instructions, the SHRDI instruction provides an adjustment to the result so the instruction can be used to divide a value by a power of two. (Normal right-shift instructions do not divide correctly when the value is negative and odd.)
- All logical operations of two operands are provided (AND, NOTAND, ANDNOT, NOR).
- An extensive set of bit instructions exists (SET BIT, CLEAR BIT, NOT BIT, SCAN a register for the first 0, or 1, BRANCH on bits set or clear), as well as instructions for accessing bit fields (MODIFY, EXTRACT).
- Single-instruction COMPARE_AND_BRANCH encodings optimize code density for these frequently executed operations.
- Conditional compare (CONCOMP) instructions speed bounds checking.

Table 1 on the next page summarizes the 80960 core architecture instruction set.

The architecture directly supports integer (signed) and ordinal (unsigned) data types. Bits, bit fields, bytes, short words, words, and double words can be manipulated in registers and transferred to and from memory. Triple words and quad words can also move between the registers and memory.

Register operations

Most 80960 instructions operate on registers. The architecture provides arithmetic, logical, bit and fit field, data movement, and comparison operations. To take full advantage of the large register set, three-operand instructions specify any register as one or both sources and/or the destination of an operation.

Arithmetic and logical. The architecture supports both standard and extended arithmetic operations. Add, subtract, multiply, and divide operations are available on 32-bit integers and ordinals. Extended multiply operates on two 32-bit ordinals and generates a 64-bit result. Extended divide divides a 64-bit ordinal by a 32-bit ordinal, producing a 32-bit quotient and 32-bit remainder. Addition and subtraction with carry allow 32-bit ordinals to provide extended precision adds and subtracts.

Table 1.
80960 instruction summary.

REGISTER OPERATIONS

ARITHMETICS

add[i | o]	Add
addc	Add with Carry
sub[i | o]	Subtract
subc	Subtract with Borrow
mul[i | o]	Multiply
emul	Extended Multiply
div[i | o]	Divide
ediv	Extended Divide
rem[i | o]	Remainder
modi	Modulo Integer
sh[lo | ro | li | ri | di]	Shift

MOVEMENT

mov[l | t | q]	Move registers to registers
lda	Load Address

COMPARISON

cmp[i | o]	Compare
cmpdec[i | o]	Compare and Decrement
cmpinc[i | o]	Compare and Increment
concmp [i | o]	Conditional Compare
test [*]	Test for condition
scanbyte	Scan for matching byte

LOGICAL

and	dst := src1 & src2
andnot	dst := src2 & (~src1)
notand	dst := (~src2) & src1
nand	dst := ~(src2 & src 1)
or	dst := src1 | src2
nor	dst := ~(src2 | src1)
ornot	dst := src2 | (~src1)
notor	dst:= (~src2) | src1
xnor	dst:= (src2 | src1) & ~(src2 & src1)
xor	dst := ~(src2 | src1) | (src2 & src1)
not	dst := ~src
rotate	Rotate Bits

BIT AND BIT FIELD

setbit	Set a Bit
clrbit	Clear a Bit
notbit	Toggle (invert) a Bit
chkbit	Check a Bit and set condition code
alterbit	Change a Bit according to an operand
scanbit	Search src for most significant set bit
spanbit	Search src for most significant cleared bit
extract	Extract specified bit pattern from a word
modify	Modify selected bits in dst with src

CONTROL OPERATIONS

BRANCH

b	Branch (± 2 MByte relative offset)
bx	Branch Extended (32-Bit Indirect Branch)
bal[x]	Branch and Link ("RISC Branch")
b[*]	Branch on Condition
cmpib[*]	Compare Integer and Branch on Condition
cmpob[*]	Compare Ordinal and Branch on Condition

FAULT

fault[*]	Fault on Condition
syncf	Synchronize Faults

PROCEDURE

call	Procedure Call (± 2 MByte relative offset)
callx	Call Extended (32-Bit Indirect Call)
calls	System Procedure Call
ret	Return

ENVIRONMENT

modpc	Modify Process Controls
modac	Modify Arithmetic Controls
modtc	Modify Trace Controls
flushregs	Flush Local Register Cache to Memory

DEBUG

mark	Conditionally generate Trace Fault
fmark	Unconditionally generate Trace Fault

MEMORY OPERATIONS

LOAD/STORE

ld[b | s | l | t | q]	Load
st[b | s | l | t | q]	Store

READ/MODIFY/WRIITE

atadd	Atomic Add (Locked RMW Cycles)
atmod	Atomic Modify (Locked RMW Cycles)

i = integer, o = ordinal, b = byte, s = short, w = word (32-bits), l = long, t = triple, q = quad,
lo = left ordinal, li = left integer, ro = right ordinal, ri = right integer, di = right dividing integer
dst = destination, src = source, x = extended,
* = Conditions: If [equal | not equal | less | less or equal | greater | greater or equal | ordered | unordered]

Arithmetic shift operations support 32-bit ordinals. Logical shift instructions operate on 32-bit integers, and a 32-bit register value can also be rotated. In addition, all possible two-operand, bitwise Boolean operations exist: AND, NOTAND, ANDNOT, XOR, OR NOR, XNOR, NOT, NOTOR, ORNOT, and NAND.

Bit and bit field. Bit operations allow bits in the registers to be set, cleared, toggled, and moved to or from the condition codes. SCAN and SPAN operations provide ways to find the most significant set or cleared bit in a register.

The 80960 contains two bit field instructions, EXTRACT and MODIFY. The EXTRACT instruction shifts a bit field in a register to the right and fills in the bits to the left of the bit field with zeros. The MODIFY instruction moves a specified bit field from one register to another when no adjustment change in bit position is required.

Data movement. A set of data movement (MOV) instructions allows register values to be copied to other registers. The MOV instructions can move from one to four registers at once. The load and store operations described later move data to and from memory.

Comparison. These instructions compare operands and set the resulting condition codes in the arithmetic controls register (Figure 3). The 80960's condition codes listed in Table 2 provide the arithmetic flag function of other architectures, in a way that allows maximum parallel execution.

In general, only compare instructions set the 80960's condition codes and conditional instructions use them. To perform an ADD followed by a conditional branch when the result is zero, a Compare_and_Branch instruction must be executed after the ADD because arithmetic instructions do not alter the condition codes.

Table 2.
Condition code encodings.

Condition code	Condition
000	Unordered *
001	Greater Than
010	Equal (True)
011	Greater Than or Equal
100	Less Than
101	Not Equal (False)
110	Less Than or Equal
111	Ordered

* Used with floating-point data types.

Although not generally noticed in a sequential execution environment, a parallel environment mandates the decoupling of the condition codes from the instruction set. The 80960 allows multiple instructions to execute simultaneously, thus giving ambiguous meaning to a set of condition codes that are altered by multiple arithmetic instructions in the same clock cycle. The 80960 approach separates condition checking and decision making from all other instructions to provide flexibility in reordering instructions for parallel execution.

The 80960 compare instructions compare both integers and ordinals. A subset of the compare instructions increments or decrements an operand after the comparison.

Memory operations

The load/store nature of the architecture decouples memory references from instruction execution. Register set and memory instructions can be executed simultaneously. Since the load data may take some time to

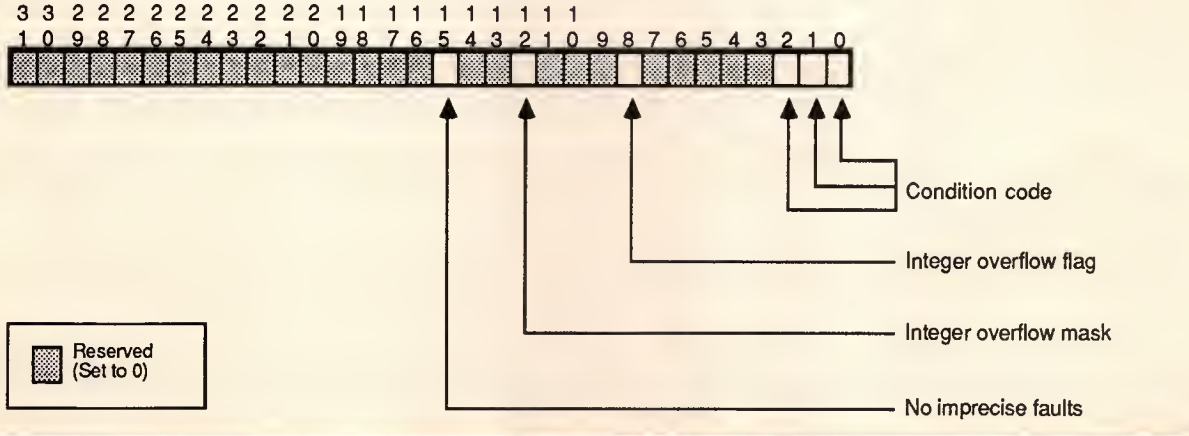


Figure 3. Arithmetic controls register.

arrive at the CPU, the load requests can be advanced in the instruction stream to overlap memory access time with other data-independent CPU operations.

Load/store. The load and store instructions copy bytes, short words, words, or multiple words to or from memory and registers. When a load integer is specified for an 8-bit or 16-bit quantity, the CPU extends the data's sign to fill 32 bits before writing the destination register. When a load ordinal is specified for an 8-bit or 16-bit quantity, the CPU attaches leading zeros to the data to fill 32 bits before writing the destination register. The store instructions allow the destination data width to be a byte, short word, word, or multiple words. When a store byte, or short word is performed, the CPU automatically formats the data being written according to the data type (integer or ordinal).

Addressing modes. The architecture supports 11 addressing modes for memory operations, as summarized in Table 3. The addressing modes selected for support provide a broad range of most-often-used simple modes. We chose a rich set of addressing modes to allow optimization for code density as well as speed.

Literals are immediate 5-bit numbers that can range from 0 to 31. Literals may be used as operands in any register operation.

The *Register* address mode is used when an operand specifies a register number (g0, r5).

The *Absolute Offset* address mode specifies the absolute address of the target as an offset from the current instruction pointer. The offset is encoded in the memory instruction opcode. If the offset is outside the range of 0 to 2048, the assembler generates a two-word instruction in which the second word is a 32-bit displacement.

Register Indirect addressing allows the address of the target to be specified by the contents of a register. An immediate offset or displacement can be added to the register to form the effective address. An index (scaled by 2, 4, 8, or 16) may also be added.

Memory operations can also specify target addresses relative to the instruction pointer, a capability useful in creating relocatable data and code.

Atomic memory operations. Two atomic memory operations support multiprocessing environments with more than one processor accessing the same memory, atomic add (ATADD) and atomic modify (ATMOD). The ATADD instruction causes an operand to be added to the value in the specified memory location. The ATMOD causes bits in the specified memory location to be modified under control of a mask. These instructions perform their memory-to-memory, read-modify-write operations with a locked bus to prevent data corruption.

Control operations

Control operations include those instructions that could result in the redirection of program flow. CALL, RET, BRANCH, and COMPARE_AND_BRANCH instructions fall into this category.

Procedure calls. The CALL instruction causes the local registers to be preserved and redirects program flow to a point indicated by an offset encoded in the instruction. The Call Extended (CALLX) instruction dif-

Table 3. Addressing modes.

Mode	Description	Assembler Example
Literal	value	0x12
Register	register	r6
Absolute offset	offset	Label + 3
Register Indirect	abase	(r6)
Register Indirect with offset	abase + offset	Label + 3 (r6)
Register Indirect with index	abase + (index • scale)	(r6) [r7 * 4]
Register Indirect with index and displacement	abase + (index • scale) + displacement	Label + 3 (r6) [r7 * 4]
Index with displacement	(index • scale) + displacement	Label [r6 * 4]
IP with displacement	IP + displacement + 8	Label (IP)

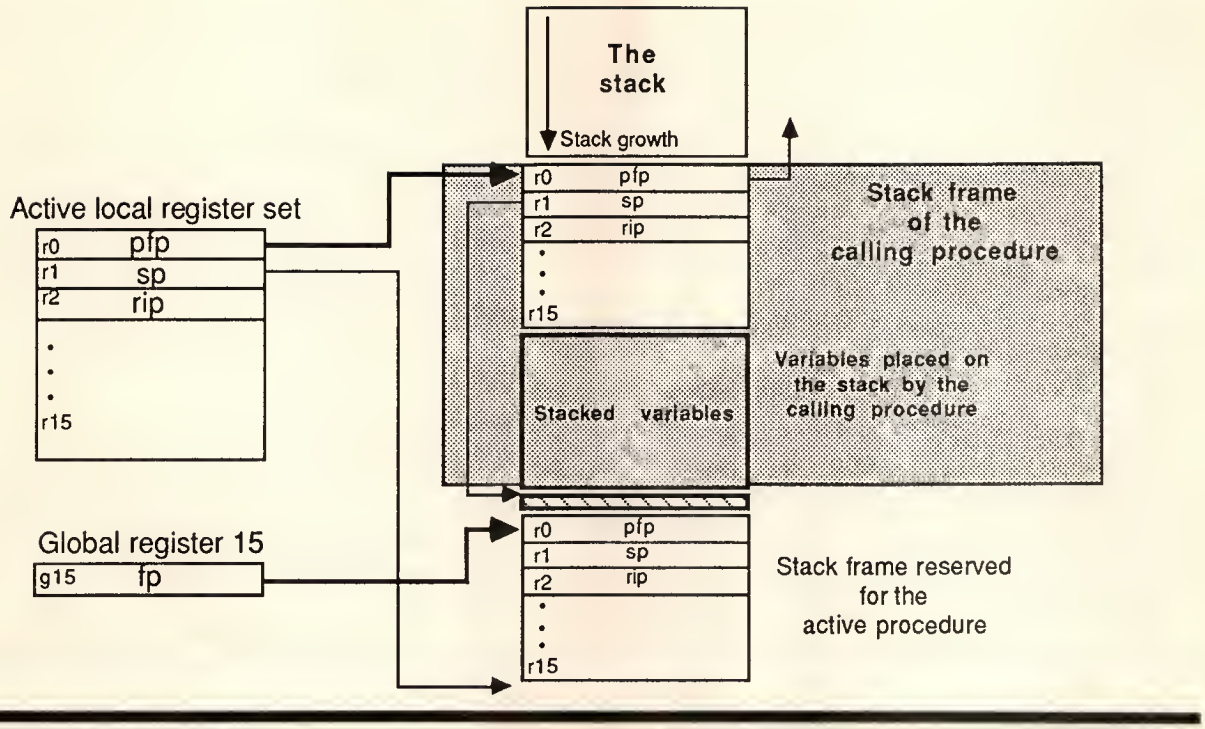


Figure 4. Procedure stack structure.

fers in that it allows a 32-bit value to provide the CALL destination. The destination can either be encoded in the instruction or specified by a register value (for example, indirect call). The Call System (CALLS) instruction gets its target address from a table of system procedure addresses explained later. The Return (RET) instruction returns control to the calling procedure and restores the local registers of the calling procedure.

The semantics of the CALL/RET allow an optimization known as register caching. A register cache keeps the context (local registers) of the most recently executing subroutines on chip so that CALL/RET instructions do not have to access memory to save or restore the local registers.

When a CALL instruction executes, the 80960 allocates a new set of 16 local registers from a pool of register sets for the called procedure. If the pool is depleted, a new register set is allocated by taking one associated with an earlier procedure and saving it in memory. A RET instruction causes the most recently cached local register set to be restored, freeing a register cache location.

The register cache contributes to performance in four ways:

- It significantly reduces the saving and restoring of registers that are usually performed when crossing subroutine boundaries.
- Since the local register sets are mapped into the stack frames, the linkage information that normally appears in stack frames (pointer to previous frame, saved instruction pointer) is contained in the local registers. Most call and return instructions execute without causing any references to off-chip memory.
- It allows compilers to map most or all of a procedure's local variables directly into registers.
- It provides a large number of registers (16 local and 16 global), which can be exploited by optimizing compilers.

The procedure stack (see Figure 4) reserves space for the cached registers of suspended procedures. When a register set must be flushed from the register cache to memory, it moves to the reserved stack frame space.

When a new procedure is entered, the 80960 allocates space for the procedure's register set as the only contents of its stack frame, although no memory accesses will occur unless the register cache is full. If the procedure desires more space on the stack for autovariables or parameter passing, it adjusts the stack pointer to reserve as much space as it needs. The procedure can then access this space using stack pointer relative addressing so long as the procedure is active. When the procedure returns, its stack is automatically reclaimed.

Branch_and_Link (BAL) performs a procedure call without saving the local registers. The 80960 saves the return instruction pointer in a global register and redirects program flow. To return from a routine that is invoked by a BAL, a BX (Branch Extended) is performed. BAL allows fast subroutine calls to leaf procedures without allocating (and possibly displacing) a new register set. Since a leaf procedure calls no other procedure, its registers can be allocated out of those remaining in the current set.

Branching. Advanced architectures have yet to deal cleanly with the dreaded branch, although some existing methods try and minimize the instruction pipeline breaks caused by branches and conditional branches. One method used by other architectures is a delayed branch. This method requires that a valid instruction *always* be placed after every branch. The delayed branch mechanism exposes the pipeline to the programmer and makes it easy to write code that "breaks." Compilers also have a difficult time finding useful instructions to *always* fill the blank pipeline slots following a branch and insert NOPs about 30 percent of the time. Furthermore, in architectures with a delayed branch mechanism, microarchitectures will be constrained in their enhancement choices.

The 80960 alternative to a delayed branch hides the pipeline and microarchitecture implementation from the programmer and allows transparent performance enhancements. For example, an 80960 microarchitecture that detects branches ahead of the executing code could fetch the branch destination to keep the pipeline full. In essence, "branch lookahead" allows branches to be executed in zero clock cycles.

Branches can be unconditional or conditional. The Branch and Branch Extended instructions perform unconditional redirection of program flow without linkage. Branch and Branch Extended differ in the width of the target address offset provided. The Branch instruction includes an encoded offset in the one-word instruction (MEMA format), whereas Branch Extended branches to the location pointed to by a register or an encoded 32-bit displacement (MEMB format).

The conditional branches use the condition codes in the arithmetic controls register to determine whether or not to take the branch. The 80960 provides all combinations of branch conditions.

Branch lookahead works well with unconditional branches but would be of marginal benefit on conditional branches since the branch target, or the instruction after the branch, cannot be executed until after evaluation of the branch condition. Pipeline breaks would, therefore, be inevitable even if the microarchitecture implemented some sort of hardware prediction mechanism. To reduce the effect of the conditional branch on performance, the 80960 defines two types of conditional branches, those that are usually taken and those that aren't usually taken. The implementation can then guess which way the branch is going to go, based upon an excellent resource capable of prediction—the programmer. Only in the case of a programmer's wrong prediction would a pipeline stall occur. Furthermore, compilers will take advantage of branch prediction when they detect loops.

It is either obvious, or uncertain, at the time the program is written which way the branches will branch most often during execution. If the likely branch outcome is obvious, the type of branch to use will be obvious. In the cases where runtime factors determine the branch path, the branch types can be selected to reduce the time through the longest path or to reduce the average path time.

Compare and branch. The compare and branch instructions support integers and ordinals. The CPU compares two operands; the result determines the branch taken. This frequently used operation is one instruction that increases performance and improves code density. The 80960 provides all combinations of branch conditions, in addition to branch-on-bit instructions.

Instruction cache

As processors increase in speed, the traffic between processor and memory becomes a significant performance bottleneck. To effectively reduce this bottleneck, we incorporated an instruction cache within the processor.

An on-chip instruction cache is highly desirable for two reasons. Caching instructions on chip greatly reduces system bus loading and the criticality of the system's memory access speed in a parallel execution environment. However, an instruction cache plays an additional role. The only way to cause multiple instructions to execute simultaneously is to decode multiple instructions simultaneously. An on-chip instruction cache gives instruction decode the capability of looking downstream and decoding and dispatching multiple instructions simultaneously for parallel execution.

The advantage of an instruction cache over a prefetch queue, a technique used in most high-performance microprocessors to date, is that a queue does not reduce the memory traffic for instructions. It only attempts to distribute the traffic more efficiently. A cache's most obvious effect occurs with execution loops, common in embedded control applications. After a loop is first executed, successive iterations of the loop generate no memory references for instruction fetches. Likewise, when a small, low-level procedure concludes and executes a RET instruction, the code for the high-level routine to which it is returning likely still resides in the cache. Thus, we reduce the sensitivity of instruction execution speed to slow memory and free valuable bus bandwidth for other operations.

Having an instruction cache requires special considerations in applications that employ self-modifying code or uploadable programs. In general, embedded applications are unaffected. However, for 80960 chips targeted at embedded applications in which volatile code exists, we will provide implementation-specific cache features. For example, implementations could provide a bus input pin that prohibits the data being read from being cached, a method for flushing the cache, a transparent instruction cache, a cache disable bit, or some other feature tuned to the application.

To allow implementations of the 80960 latitude in the amount and type of cache provided, the architecture does not specify the instruction cache parameterization.

User-supervisor protection

The architecture provides a mode and stack switching mechanism called the user-supervisor protection model. This protection model allows a system design in which the kernel code and data reside in the same address space as the user code and data. However, the access to the kernel procedures (called supervisor procedures) occurs through a specified interface. A data structure called the System Procedure Table provides this interface (Figure 5).

The 80960 references the System Procedure Table when a System Call (CALLS) instruction executes. This call is similar to a local call, except that the processor gets the location of the called procedure from the System Procedure Table. Figure 6 illustrates the use of the CALLS instruction. CALLS requires a procedure-number operand that is used as an index into the table.

The System Procedure Table entry referenced by CALLS specifies an entry pointer and an entry type for the called procedure. The 80960 invokes two types of system procedures, *local* and *supervisor*. A procedure that is specified as a local procedure is invoked as if it were called by the CALL or CALLX instructions, except that the processor gets the entry point of the called procedure from the System Procedure Table. Referencing a supervisor procedure, on the other hand, switches the processor to the supervisor execution mode and to the supervisor stack.

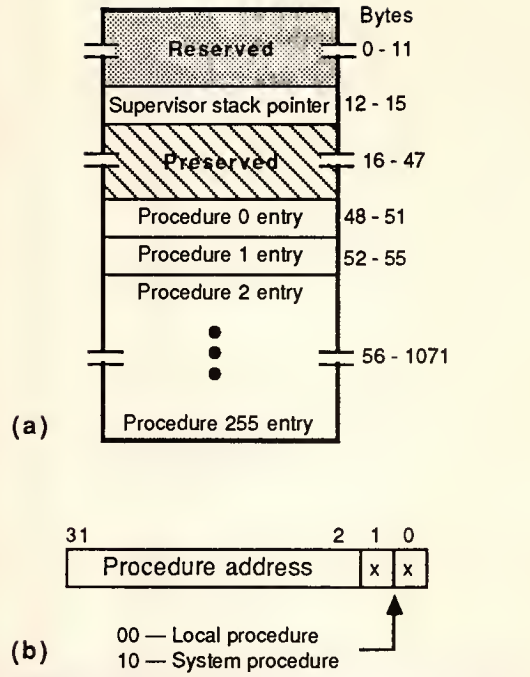


Figure 5. Structure of the System Procedure Table (a) and a procedure entry (b).

Real-time kernel procedures in the supervisor mode execute using a different stack than the one used to execute application programs procedures. Special, supervisor-only instructions also execute in supervisor mode. The MODPC instruction (used to change the processor priority) is always a supervisor instruction. Instruction set extensions that control on-chip hardware are also likely to be restricted to the supervisor mode.

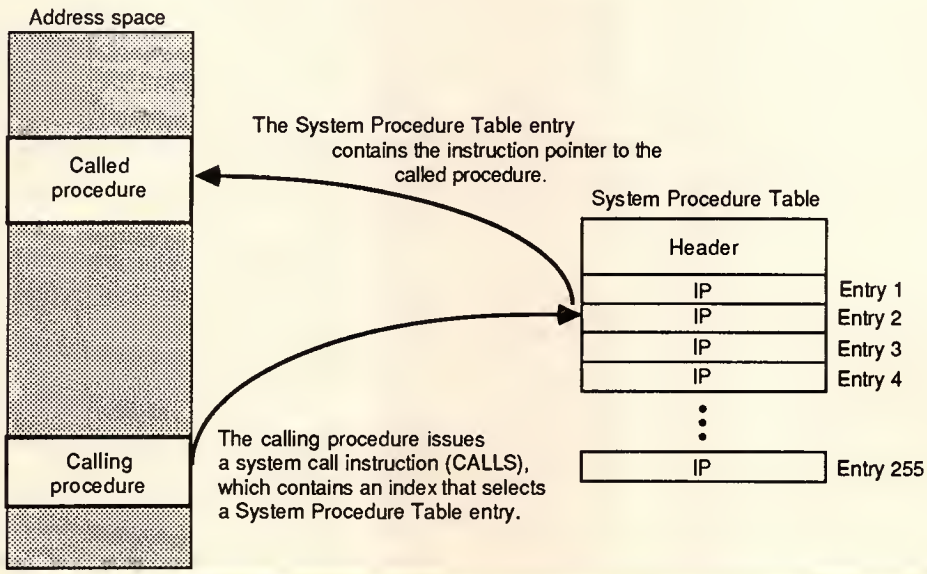


Figure 6. Example of a system procedure call.

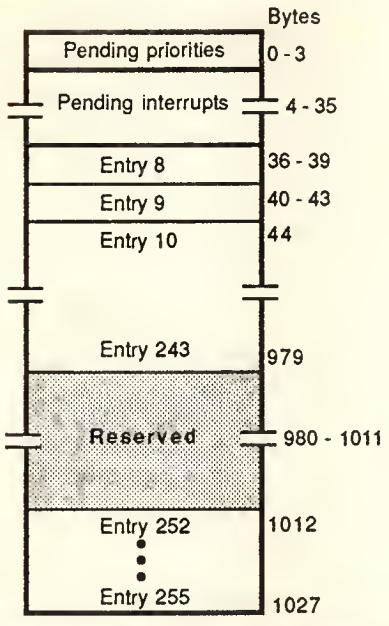


Figure 7. Structure of the interrupt table.

The processor remains in the supervisor mode until the procedure that caused the original mode switch performs a return. Switching stacks and protecting against stack corruption help maintain the integrity of the kernel. For example, the mechanism allows access to system debugging software or a system monitor even if the application crashes.

Interrupts

The 80960 contains a priority interrupt model and a mechanism for queueing pending interrupt requests without user program intervention. When an interrupt is signaled and its priority is higher than the current processor priority, the CPU invokes an interrupt handler and the processor priority changes to that of the interrupt. Otherwise, the 80960 saves the interrupt for service until it becomes the highest priority request pending.

The interrupt table seen in Figure 7 holds the 32-bit pending priorities field, the 256-bit pending interrupt field, and the 248 interrupt vectors. Within each processor priority the 80960 contains eight vectors, eight associated pending interrupt bits, and one pending priority bit. The pending priorities field indicates the priorities at which pending interrupts await. The pending interrupt field indicates exactly which requested interrupts have not yet been serviced.

A pending priority bit is simply the OR of all eight pending interrupt bits at a particular priority. This field optimizes checking for pending interrupts by the processor. When an interrupt request will not be serviced immediately, the 80960 sets the bit in the pending interrupt field associated with the request. It also sets the pending priorities bit associated with the priority of the request. When the running priority of the processor drops below that of the pending interrupt, the 80960 services the interrupt and clears the associated pending bit. The CPU also clears the associated pending priority bit if appropriate.

Faults

Processors use fault mechanisms to handle exceptions or errant conditions that a program may or may not be capable of correcting. We defined the 80960's fault mechanism for an environment in which parallel or out-of-order execution occurs. When a fault is generated, the processor calls the appropriate fault handler. The 80960 automatically provides the handler with an extensive set of information about the faulting condition for correction or analysis.

It is possible that when a fault is detected not enough information would exist to determine the exact instruction that faulted. For example, when multiple instructions execute in one clock cycle, multiple faults could occur in a single clock cycle. This "imprecise" condition could generate a fault that we call imprecise. A tightly coupled fault handler may be able to recover proper program execution when an imprecise fault occurs. Precise faults are those from which recovery is easy.

The 80960 provides two controls over the generation of imprecise conditions and faults. The first fault control method, a global control bit, puts the processor in a mode where no imprecise conditions are created (No Imprecise Faults, or NIF mode). In this mode, the 80960 restricts parallel execution. All faults are precise. The NIF bit can be used to create a critical region in which all faults are precise. The second fault control mechanism is the Synchronize_Faults (SYNCF) instruction. This instruction halts execution until all pending operations complete, and all faults up to that point have been signaled. It is useful on Ada block boundaries where different blocks can have different fault handlers.

An 80960 implementation detects various conditions in code or in its internal state (called fault conditions) that could cause the processor to deliver incorrect or inappropriate results, or that could cause it to take an undesirable control path. For example, the 80960 can recognize (if enabled by the user) divide-by-zero and overflow conditions on integer calculations as a fault. The architecture also recognizes inappropriate operand values and attempts to execute unimplemented opcodes, among other conditions, as faults.

When a fault is detected, the system processes it immediately and independently of the program or handler that is executing at the time. The fault mechanism is similar to that used by the interrupts. Several fault

types exist, in which the fault type determines which entry in the Fault Table (Figure 8) is invoked for a particular fault. The Fault Table contains one entry for each fault type. The entry defines a particular fault handler routine as a local procedure or a system procedure. When the fault handler is a local procedure, the Fault Table entry contains the address of the procedure entry point. When the fault handler is a system procedure, the Fault Table entry contains the system procedure number, which selects the correct entry point from the System Procedure Table described earlier.

Figure 9 describes the fault record, which is the information provided to a fault handler when a fault occurs. Table 4 on page 76 summarizes the fault types and subtypes that are currently defined in the 80960 architecture. As extensions to the architecture that consume additional fault types become available, the encoding of fault types and subtypes will occur in such a way that every implementation capable of recognizing similar faulting conditions encodes them identically. For example, the 80960KB adds the floating-point faults (fault type 4). Any other 80960 implementations that also recognize floating-point faults also encode them as fault type 4.

Debug support

Another objective of the architecture is to support software debugging and tracing. A trace-controls register enables most of this support. The trace controls detect any combination of the following events:

- Instruction execution (single step),
- Execution of a Taken Branch instruction,
- Execution of a Call instruction,
- Execution of a Return instruction,

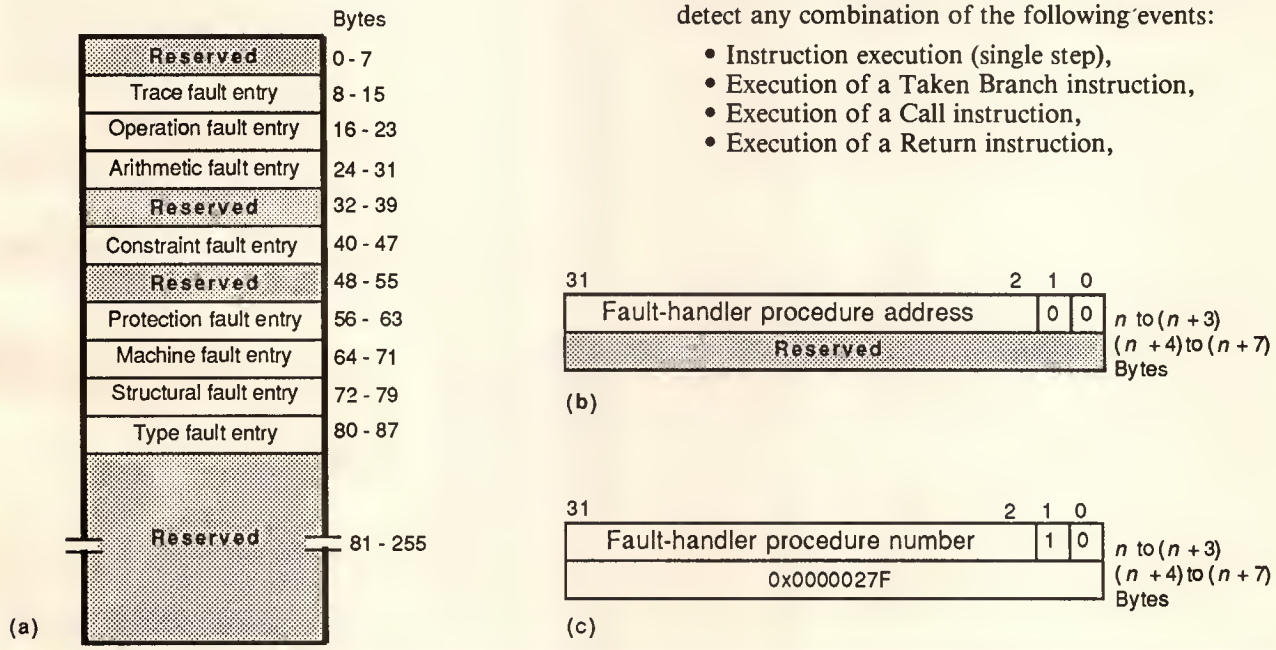


Figure 8. Structure of the fault table (a); an entry to reference a local procedure (b); and an entry to reference a system procedure (c).

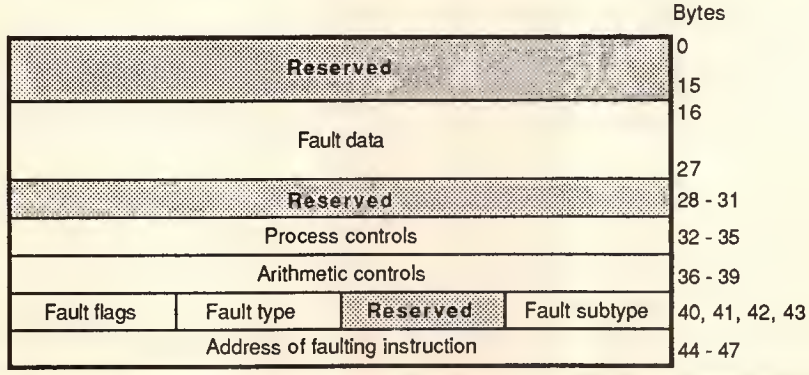


Figure 9. Fault record information. The return pointer r2 is also provided.

Table 4. Fault types and subtypes.

Fault type	Fault Subtypes	Comments
Arithmetic	Overflow, underflow	Integer overflows/ divide by zero
Constraint	Range	If FAULT_IF taken
Protection	Length	Procedure # in CALLS out of range
Machine	Bad access	Memory access failed to complete
Type	Mismatch	Tried to execute supervisor instruction in nonsupervisor mode
Operation	Invalid opcode, Invalid operand	Tried to execute invalid opcode, or an operand in a valid opcode was invalid
Trace	Instruction, branch, call, return, prereturn, supervisor, breakpoint	Trace event occurred

• Detection that the next instruction is a Return instruction,
• Execution of a supervisor or system call, and
• Breakpoint (hardware breakpoint or execution of a breakpoint instruction).

When a trace event is detected, the processor generates a trace fault to give control to a software debugger or monitor. Since all 80960 implementations are likely to have an on-chip cache, external hardware cannot trace the flow of instruction execution by monitoring the external bus. Therefore, to trace instruction execution, a debugger could enable the BRANCH, CALL, and RET trace faults and reconstruct the instruction-by-instruction flow of a program. This method, however, will not provide transparent, or real-time tracing. When noninvasive emulation is desirable, the user should employ an in-circuit emulator.

The 80960, an extensible embedded control architecture, maximizes computational and data processing speed through parallel execution. The first implementation of the architecture (80960KB) achieves single-clock execution of instructions, while fractional clock instruction rates are architecturally unhindered and will be available in future implementations.

Acknowledgments

Too many people contributed to the 80960 effort to list them here. However, I relied upon the following, either in person or through their writings, in developing this article: Dave Budde, Glen Hinton, Mike Imel, Konrad Lai, Glenford J. Myers, Lew Pacely, Fred Pollack, Rob Riches, Frank Smith, and Randy Steck.

Bibliography

80960KB Programmer's Reference Manual, No. 210567-001, Intel Corporation, Santa Clara, Calif., 1988.

Embedded Controller Handbook, Vol. 1, No. 210918-006, Intel Corporation, 1988.

ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic, IEEE Comp. Soc., Los Alamitos, Calif., 1985.

David P. Ryan, a senior applications engineer in Intel's embedded controller operation, supervises the Arizona-based 32-bit applications team. He began work on 32-bit embedded controller requirements in early 1985 and full-time work on 32-bit products in early 1987. Before this, he supported the MCS-96 family of 16-bit microcontrollers.

Ryan holds a BSEE from Arizona State University and an MBA from Pennsylvania State University.

Questions concerning this article can be addressed to the author at Intel Corporation, Embedded Controller Operation, 5000 West Chandler Boulevard, CH3-64, Chandler, AZ 85226.

A Pipelined Interface for High Floating-Point Performance with Precise Exceptions

The NS32532/32580 processor cluster teams with Weitek's WTL3164 to deliver a peak floating-point performance of 15 MFLOPS at 30 MHz without compromising exception precision.

Sorin Iacobovici
National Semiconductor

In spite of spectacular advances in very large scale integration density in recent years, today's technology still forces the computer architect to make a crucial decision. Which features belong on the central processing unit (CPU) chip and which on a coprocessor chip?

With few exceptions, architects implement floating-point arithmetic support on a coprocessor chip. One main advantage of this approach is CPU support for an external floating-point unit that requires significantly fewer transistors than does integrating the FPU on chip. The remaining transistors can increase CPU integer performance. Another advantage of the coprocessor approach is the following range of floating-point calculation-support solutions:

- emulating floating-point instructions in software, which decreases the system price by eliminating the need for an FPU chip;
- a simple, lower cost, lower performance FPU; or
- a higher cost, higher performance FPU.

On the other hand, the usual disadvantages of implementing floating-point support on a separate FPU chip are:

- lower performance due to CPU-FPU communication overhead, and/or
- higher CPU pin count to support a more efficient CPU-FPU communication protocol.

National Semiconductor's 32532/32580 processor cluster features a pipelined CPU-FPU interface that overcomes these disadvantages. This approach minimizes the CPU-FPU communication overhead.

Most existing CPU-FPU processor clusters operate on a sequential model for program execution. The CPU issues a floating-point instruction to the FPU and stops so the FPU can signal back the successful completion of the instruction. This serial approach ensures that the CPU's program counter (PC) sequences through instructions one by one, finishing one before proceeding to the next. Although acceptable for low and medium floating-point performance, the overhead of the serial approach is unacceptable for high-performance FPUs like the NS32580, which uses Weitek's WTL3164[1] chip as its floating-point data path (FPDP).

The WTL3164's throughput is the equivalent of two NS32532 clock cycles for instructions like ADD, SUBTRACT, and MULTIPLY. This throughput applies to both 32-bit (single) and 64-bit (double) precision. The WTL3164 is pipelined to achieve this performance. A serial CPU-FPU communication protocol cannot take advantage of the high FPDP throughput because no new instruction enters the pipeline before the previous one finishes.

072-1732/88/0600-0077$01.00 © 1988 IEEE

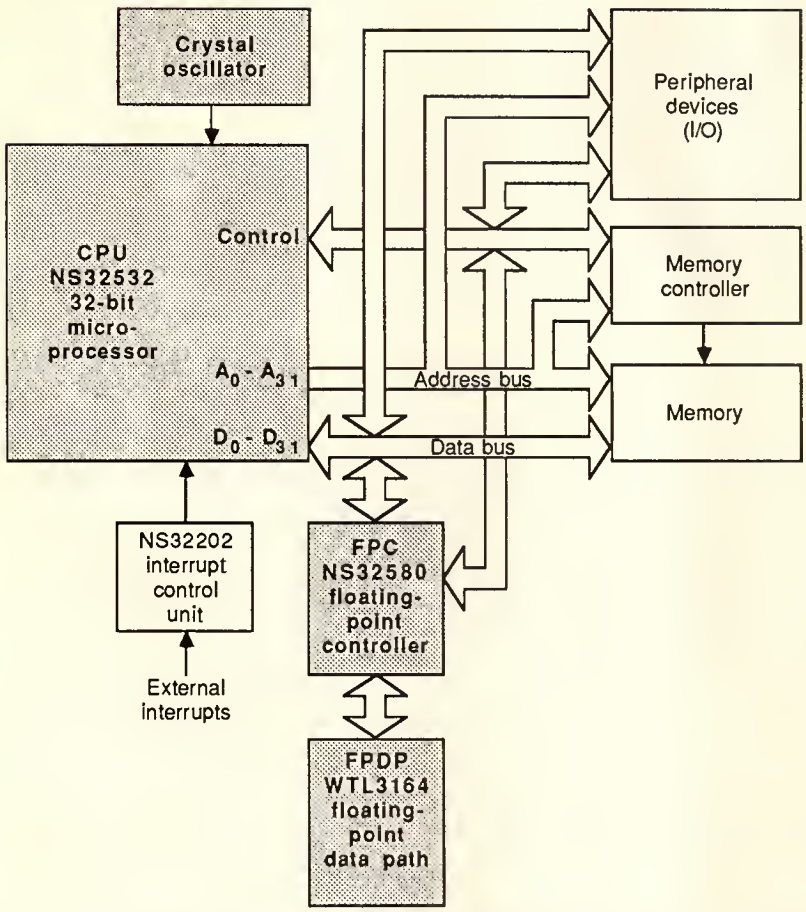


Figure 1. Simple single-processor system using the 32532/32580 cluster.

A pipelined coprocessor interface, however, does take advantage of a high FPDP throughput. Except for special cases described here, the CPU proceeds from one floating-point instruction to the next without waiting for execution completion. The floating-point instructions queued in the FPU await their turn to execute. The CPU-FPU bus, the FPU's bus interface, the FPDP, and the FPU's circuit indicating the floating-point instruction outcome (successful completion or exception) all become stages of the floating-point instruction-execution pipeline. Each FPDP stage is a separate pipeline stage.

In addition to memory, a system's state consists of

- general-purpose registers (GPRs),
- special registers (like the PC) local to the CPU, and
- floating-point registers local to the FPU.

In case of a floating-point exception, the system state saved for a pipelined floating-point instruction execution should be consistent with the system state for the sequential execution. The floating-point exceptions in this case are precise.[2] To be compatible with previous Series 32000 CPU-FPU clusters that implemented a "serial" CPU-FPU interface protocol, the 32532/32580 cluster should feature precise floating-point exceptions. Precise exceptions are also necessary for graceful recovery from arithmetic exceptions. Here, I present two options we considered for a pipelined CPU-FPU interface, as well as the CPU recovery mechanisms that provide precise floating-point exceptions for each option. The first option supports parallel execution of both floating-point and integer instructions, while the second option pipelines only the execution of floating-point instructions. The 32532 implements the second option because this alternative offers high performance with significantly lower complexity.

Floating-point architecture

The Series 32000 instruction-set architecture features basic floating-point instructions.[3] These instructions need to be executed identically by an on-chip, floating-point execution unit or by an off-chip FPU. Identical execution means identical results and behavior in case of floating-point exceptions.

The Series 32000 architecture contains eight floating-point registers for both single (32-bit) and double (64-bit) precision. They supplement the GPRs of the architecture, which are integer registers. The instruction type implies which kind of register should be used. For instance, floating-point instructions with one or more register operands imply the use of floating-point registers.

Exceptions to this rule are integer-to-floating-point and floating-point-to-integer conversion instructions. These instructions imply the use of a floating-point register for the floating-point operand and of a GPR for the integer operand. In some cases, the floating-point instruction might fail, generating one of the exceptions defined by the *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*.[4] An example is the attempt to divide a finite number by zero. The status bits in the floating-point status (FSR) register indicate the exception type. Once set by an exception, these bits stay unchanged until cleared by software. Other bits of

the FSR let the software mask each type of floating-point exception separately. An unmasked floating-point exception sets the proper FSR status bits and traps the CPU, while a masked floating-point exception only sets the corresponding FSR status bits.

The Series 32000 architecture defines a special type of interruption—Trap(FPU)—for unmasked floating-point exceptions. When such an exception occurs, the CPU saves the PC of the failed instruction and jumps to a routine pointed to by a special address in the interrupt dispatch table. The Trap(FPU) service routine can then address in software the problem that generated the exception and reexecute the floating-point instruction. These floating-point exception-handling routines are usually part of the floating-point libraries.

CPU-FPU interface

Figure 1 is a simple system block diagram of a CPU-FPU cluster that uses the CPU's data bus and special control lines for CPU-FPU communication. This approach saves CPU package pins. Such a system using the 32532 CPU[5] and the 32580/WTL3164 FPU delivers 15-MIPS peak integer performance and 15-MFLOPS peak floating-point performance.

Advanced microprocessors like the 32532 integrate caches on chip to balance the speed of their instruction execution pipeline and the memory bandwidth. The on-chip caches make many memory transactions (such as instruction fetch, operand fetch) invisible to the external circuitry, including the FPU. Thus the 32532 CPU—hereafter simply called the CPU—has to use special, two-clock-cycle, slave (coprocessor) bus cycles to transfer instructions and memory operands to the FPU (Figure 2). No CPU support is necessary for reading or writing an operand into one of the floating-point registers, since the floating-point registers are local to the FPU. The FPU plays the role of a remote, floating-point execution unit of the CPU.

A floating-point instruction execution starts when the CPU sends the instruction to the FPU. Using a slave bus cycle, the CPU actually sends a 32-bit word that encodes an identifier field to indicate that the instruction is floating point. It also encodes an opcode field that indicates the instruction to be executed. If any of the instruction's source operands are in memory, the CPU reads them and sends them to the FPU, using slave bus cycles.

At the end of the floating-point instruction execution by the FPU, the CPU is usually free to proceed to the next instruction except in the following cases:

- The floating-point instruction destination is in memory.
- The floating-point instruction changes bits (flags) in the CPU's processor status register (PSR), as in the case of FLOATING-POINT COMPARE instructions.
- A floating-point exception occurs during the floating-point instruction execution.

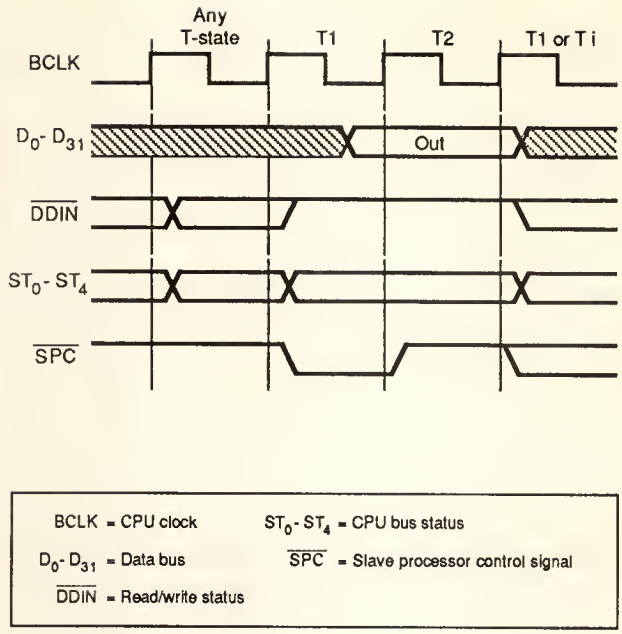


Figure 2. Slave write bus transaction.

The FPU signals the last two conditions to the CPU by asserting a special line, Force Slave Status Read (FSSR). If a floating-point exception occurs, the CPU traps with the Trap(FPU) vector. If not, the CPU updates the proper PSR bits. For all other cases, the FPU signals the (successful) completion of the floating-point instruction by asserting the Slave Done (SDN) line. If the floating-point instruction destination is one of the floating-point registers, the CPU can proceed. If the destination is in memory, the CPU reads the result from the FPU and stores it in memory.

The CPU-FPU interface protocol just discussed implements a straightforward solution that maintains the precision of floating-point exceptions. This solution consists of stalling the CPU until the outcome of the floating-point instruction execution by the FPU is known. Even when no additional CPU processing is necessary to support the current floating-point instruction, the CPU waits for the SDN signal before proceeding to the next instruction. (Examples are floating-point instructions with floating-point register destinations.) As a result, if a floating-point exception occurs, the operating system saves the proper PC and system state before it services the exception. The CPU uses this serial protocol to interface with the NS32381 FPU. (The NS32381 is a single-chip, lower cost FPU that uses a different technique from those described here for reducing CPU-FPU communication overhead and achieving good performance.)

The serial CPU-FPU interface protocol is unacceptable for very fast, pipelined FPDPs like the WTL3164.

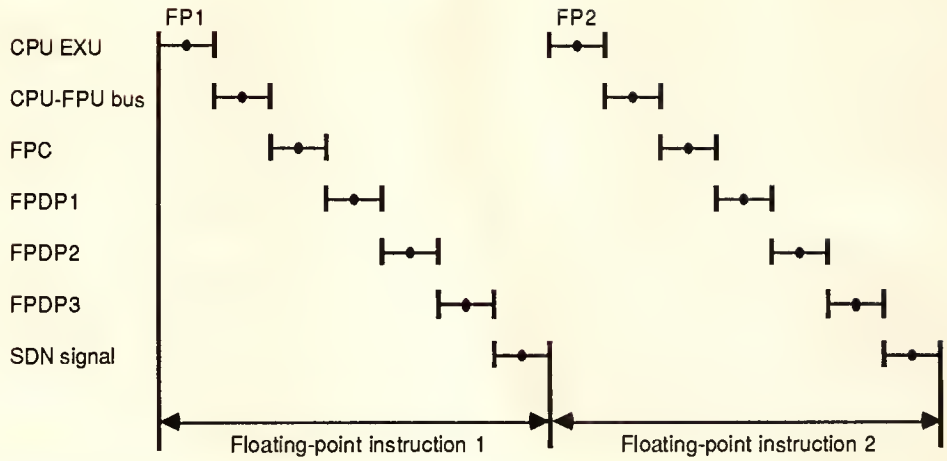


Figure 3. Serial floating-point instruction execution.

To illustrate the serial-protocol overhead problem, we use Figure 3 to represent the main execution stages of the 32532/32580 cluster, inclusive of the stages of the WTL3164 FPDP. These stages consist of:

- the CPU's execution unit (CPU EXU), which issues the floating-point instruction to the FPU and stops to wait for the SDN or FSSR handshake signals;
- the CPU-FPU bus;
- the floating-point controller (FPC), which receives the instructions and the eventual memory operands from the bus and passes them to the FPDP;
- the FPDP, which is pipelined; and
- the floating-point status-decode circuit that generates the SDN or FSSR signals based on the outcome of the floating-point operation.

As shown by Figure 3, the floating-point, instruction-execution time equals the sum of the time spent by an instruction in all the represented stages. As such, the instruction-execution time takes 14 CPU clock cycles. For the very fast FPDP used by the FPU (throughput of two CPU clock cycles per instruction), the CPU-FPU cluster speed represents only 14 percent of the FPDP speed. The protocol overhead in this case is 600 percent. This high overhead results from the fact that the CPU waits for an SDN signal before proceeding to the next instruction. This serial protocol provides a simple, though inefficient, way to achieve precise floating-point exceptions.

Pipelined CPU-FPU interface

Another method can reduce protocol overhead: letting the CPU's EXU proceed to the next instruction while the FPU buffers the issued instructions until the FPDP is available (Figure 4). However, if no recovery mechanisms are present, the CPU execution of new instructions could change the system state. This change results in imprecise floating-point exceptions that are therefore incompatible with Series 32000 architecture. To prevent this, the CPU's EXU proceeds to the next instruction only for those floating-point instructions that modify registers provided with a recovery mechanism. The floating-point instructions that qualify

- do not have memory destinations (for example, the destination is a floating-point register local to the FPU);
- do not modify the CPU's PSR (it is not a FLOATING-POINT COMPARE instruction); or
- have memory operands that do not use addressing modes like Top of Stack (TOS) that modify a CPU register.

The CPU uses a simple recovery mechanism—the floating-point instruction FIFO buffer (FIF)—to maintain the precision of floating-point exceptions. This mechanism consists of a FIFO buffer that saves any CPU registers that can be changed. The CPU saves the registers in the FIF when the CPU issues the corresponding floating-point instruction to the FPU (Figure 5).

The definition of the SDN and FSSR signals for pipelined slave protocol remains the same as for the nonpipelined protocol described previously. An SDN represents successful floating-point instruction completion, while FSSR usually indicates the need for CPU intervention, as in the case of an unmasked floating-point exception. The FPU asserts these signals in the same order in which the CPU issued corresponding floating-point instructions to the FPU. An SDN signal received by the CPU simply advances the FIF. As a

result—and this is important to note—the PC and other register values at the FIF output correspond to the floating-point instruction currently being executed by the FPU.

Therefore, when the executing floating-point instruction signals an exception to the CPU by asserting the FSSR signal, the CPU replaces the modified registers with their original values saved in the FIF and flushes any remaining FIF entries. In this way, when the trap routine starts executing, the system state is consistent with the sequential architecture model (precise floating-point exception).

We studied two pipelined slave-interface options for the CPU: one that allows parallel execution (overlapping) of floating-point and integer instructions, and one that allows parallel execution of floating-point instructions only.

Overlapping the floating-point instructions and integer instruction execution holds the promise of higher performance. However, this overlapping requires implementation of relatively complex recovery mechanisms inside the CPU to maintain the precision of floating-point exceptions.

When allowing parallel execution of floating-point and integer instructions, the CPU saves not only the PC in the FIF, but also the CPU's PSR (Figure 6). Saving the PSR is necessary because some of the integer instructions modify PSR bits, and the PSR has to be restored to its original value in case of a floating-point exception.

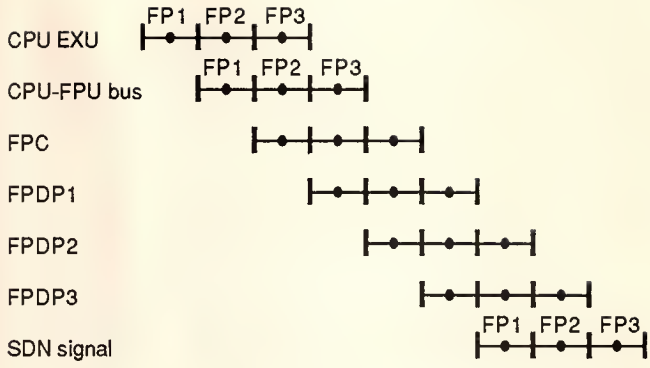


Figure 4. Pipelined floating-point instruction execution that reduces the CPU-FPU interface protocol overhead.

As shown in Figure 7, the CPU can proceed to the next instruction after issuing the floating-point instruction to the FPU. Exceptions to this rule are the special cases discussed later. Figure 4, shown above, represents the pipelining of the floating-point instruction execution over the CPU's EXU, the CPU-FPU bus, the FPC, and the FPDP. As long as no special case occurs, the CPU-FPU cluster delivers a peak throughput of two CPU clock cycles for register-to-register, floating-point instructions—equal to the FPDP throughput (no protocol overhead).

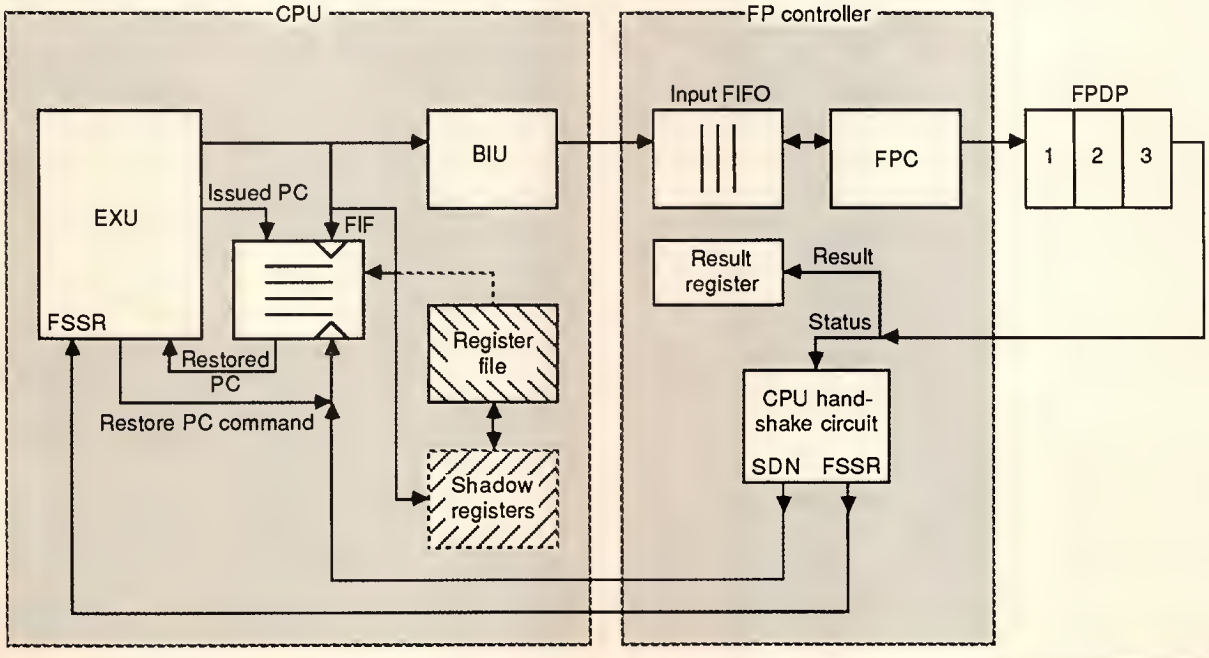


Figure 5. Recovery mechanisms for maintaining the precision of floating-point exceptions.

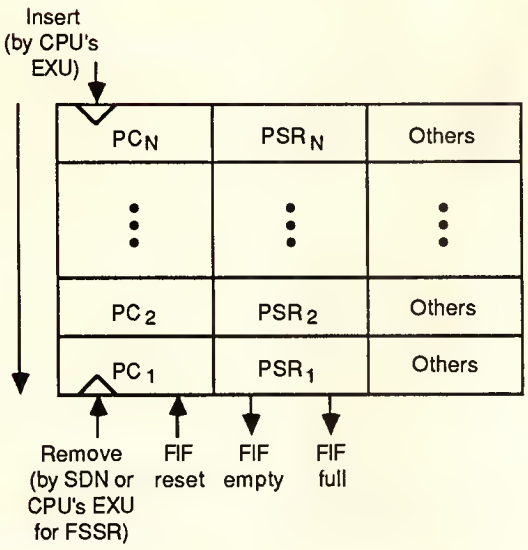


Figure 6. FIF configuration that supports floating-point and integer instruction-execution overlap.

If the floating-point instruction destination is in memory, the CPU sends the instruction and operands to the FPU and stops to wait for the FIF to empty. "FIF empty" indicates here that all floating-point instructions sent to the FPU completed successfully. It also indicates that the result of the last one is available for the CPU to read and store in its memory destination. This approach prevents out-of-order memory modification, while maximizing performance. The CPU sends the opcode and eventual memory operands to the FPU overlapped with FPU execution of previous floating-point instructions. If the next integer instruction has a memory destination, the CPU has to detect this and stop to wait for the FIF empty condition. It does this to prevent memory modification before confirmation that no floating-point exceptions have occurred.

Nothing fundamental prevents pipelining the floating-point instructions with memory destinations. We made the decision to serialize the floating-point instruction execution when executing such instructions to simplify the CPU design.

If integer instructions with destinations in the CPU registers follow a floating-point instruction with a destination in a floating-point register, the CPU can start executing integer instructions immediately after issuing the floating-point instruction to the FPU. To do this and still maintain the precision of floating-point exceptions, the CPU must implement shadow registers as shown in Figure 5. These registers store the old values of the CPU registers before they are modified. In case of a floating-point exception, the CPU restores its register values using the values in the shadow registers. As a result, the system state remains consistent with the sequential architecture model.

A floating-point instruction like FLOATING-POINT COMPARE, which modifies the CPU's PSR, signals completion to the CPU by asserting the FSSR line. As the FSSR line requests CPU intervention, the CPU sends the instruction and operands to the FPU and stops to wait for the FSSR.

The CPU must also detect and treat carefully the TOS addressing mode that implicitly changes the CPU's stack pointer (SP) register. A possible approach to meeting this requirement is saving the SP register in the FIF and restoring it in case of a floating-point exception. However, the performance return of allowing the floating-point instructions with TOS operands to be pipelined does not justify this FIF complication. A simpler approach is to have the CPU detect the floating-point instructions with TOS operands and issue them to the FPU. This procedure should occur only after the FIF becomes empty (successful completion of all previous floating-point instructions), as detailed in Figure 7.

The previous description makes it clear that the CPU implements a recovery mechanism to maintain floating-point exception precision when using the pipelined slave interface. To support this mechanism, the CPU detects and correctly treats floating-point instructions that require special precautions. Because of this, the floating-point instructions are less generic from the CPU point of view than they are in the case of the non-pipelined slave interface. However, this is a small price to pay for the high performance improvement that results from the pipelined slave interface.

Figure 8 details FPU support for floating-point instruction execution when using the pipelined slave interface. The FPU passes the instructions and their eventual memory operands to the FPDP. If the FPDP is busy with longer instructions like DIVIDE or SQUARE-ROOT, the FPU saves the instructions and operands in its input FIFO. After finishing the execution of an instruction, the FPU decodes the FPDP status and sends the proper handshake signal to the CPU. It sends signal SDN if the floating-point instruction completes successfully and is not a FLOATING-POINT COMPARE instruction. If the instruction fails, generating a floating-point exception, or if it is a FLOATING-POINT COMPARE instruction that modifies the CPU's PSR, the FPU sends an FSSR signal to the CPU.

When executing a floating-point instruction with a memory destination, the FPU stores the result in a special Result register and issues an SDN signal to the CPU. Then it stops to wait for the CPU to read the result and store it in memory.

To understand the performance contribution of each of the described mechanisms, we analyzed the Whetstone benchmark compiled with our CTP Fortran compiler. We wanted to understand how overlapping the execution of floating-point instructions and integer instructions contributed to performance. We classified the groups of floating-point instructions that can be pipelined into

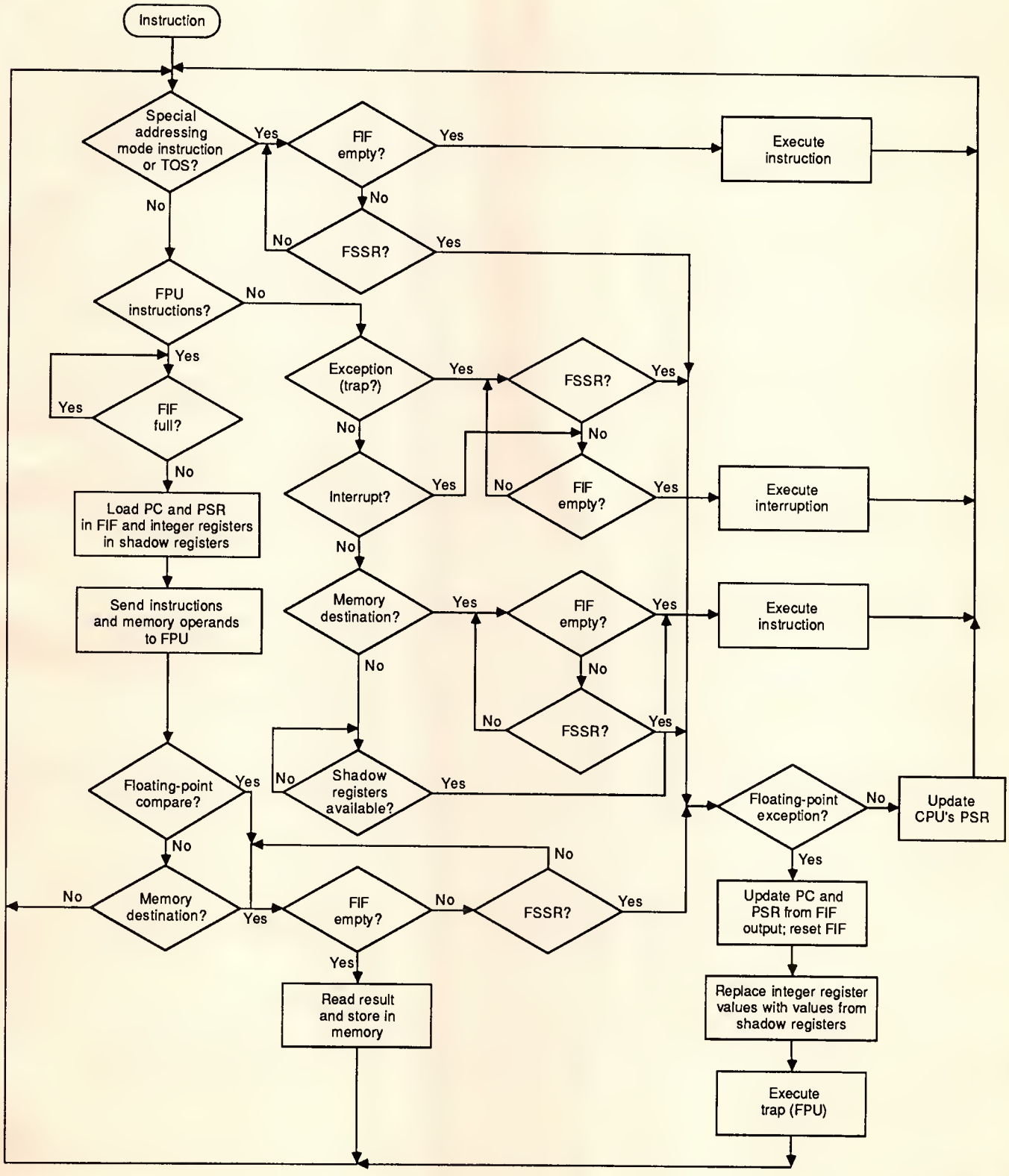


Figure 7. CPU instruction execution for pipelined slave interface with floating-point and integer instruction-execution overlap.

• groups that end in a floating-point instruction having the destination operand in memory, and
• groups that end with an integer instruction having a CPU register destination, etc.

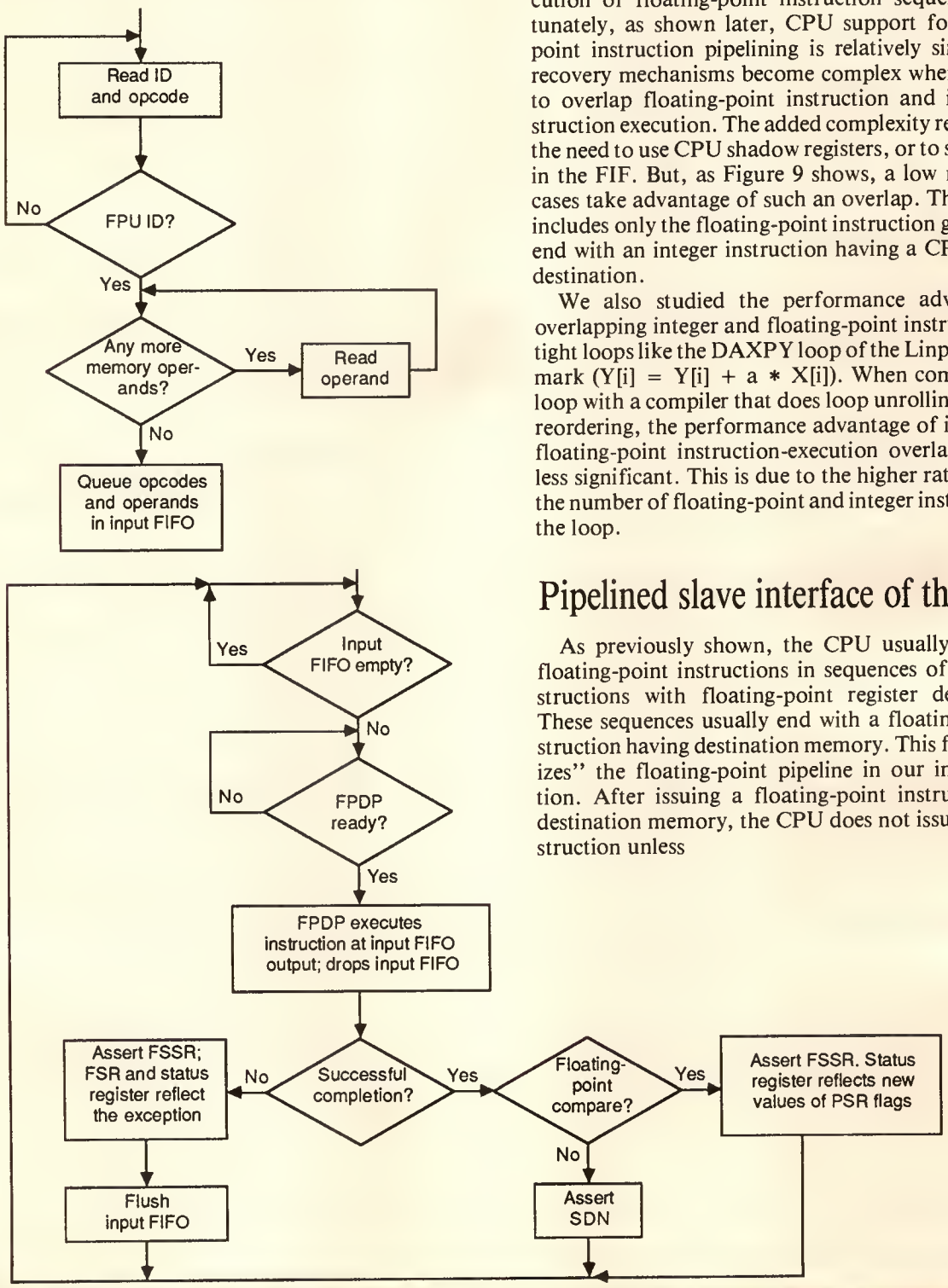


Figure 8. FPU queueing and execution of floating-point instructions.

Figure 9 represents the results of the analysis. The instruction distribution in this figure shows that the most significant contribution to floating-point performance improvement is due to pipelining (overlapping) the execution of floating-point instruction sequences. Fortunately, as shown later, CPU support for floating-point instruction pipelining is relatively simple. The recovery mechanisms become complex when one tries to overlap floating-point instruction and integer instruction execution. The added complexity results from the need to use CPU shadow registers, or to save a PSR in the FIF. But, as Figure 9 shows, a low number of cases take advantage of such an overlap. This number includes only the floating-point instruction groups that end with an integer instruction having a CPU register destination.

We also studied the performance advantage of overlapping integer and floating-point instructions for tight loops like the DAXPY loop of the Linpack benchmark (Y[i] = Y[i] + a * X[i]). When compiling this loop with a compiler that does loop unrolling and code reordering, the performance advantage of integer and floating-point instruction-execution overlap becomes less significant. This is due to the higher ratio between the number of floating-point and integer instructions in the loop.

Pipelined slave interface of the 32532

As previously shown, the CPU usually issues the floating-point instructions in sequences of several instructions with floating-point register destinations. These sequences usually end with a floating-point instruction having destination memory. This fact "serializes" the floating-point pipeline in our implementation. After issuing a floating-point instruction with destination memory, the CPU does not issue a new instruction unless

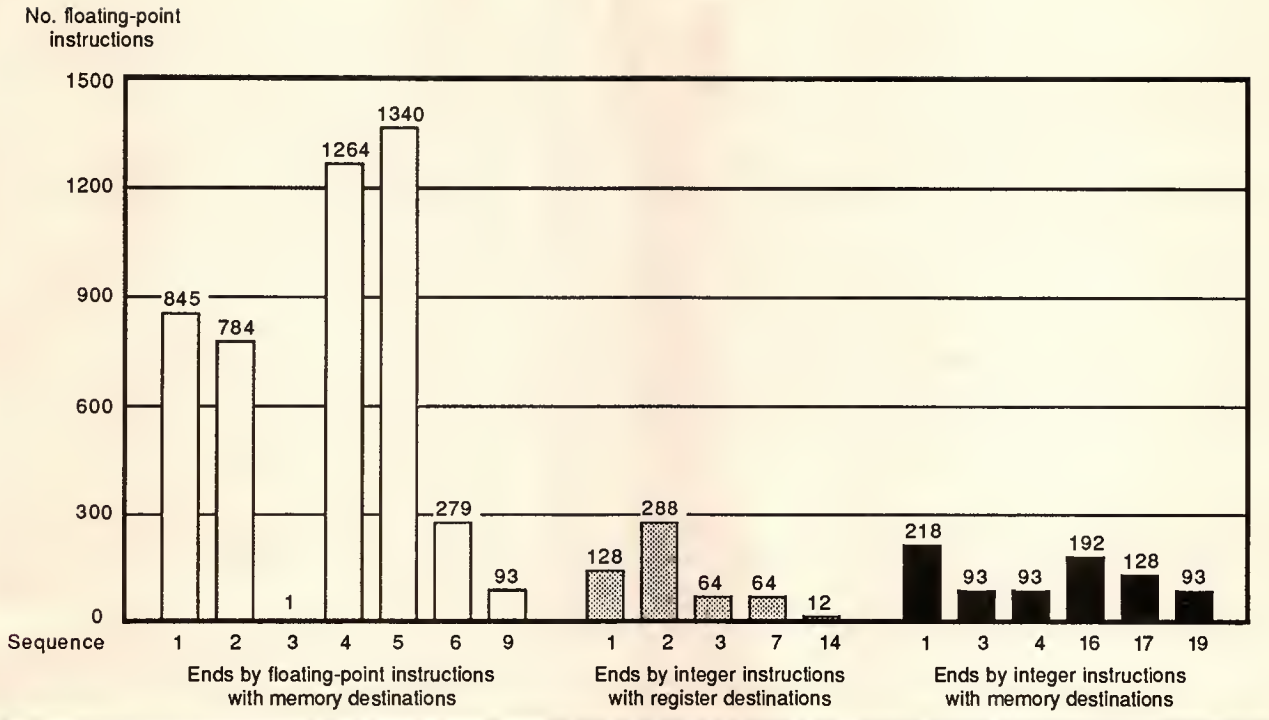


Figure 9. Analysis of floating-point instruction sequences in the Whetstone benchmark.

- all previously issued instructions successfully complete, and
- the result of the last instruction is read from the FPU and stored in memory.

Only rarely does an integer instruction—or a serializing floating-point instruction—end such sequences. Examples of serializing floating-point instructions are FLOATING-POINT COMPARE or a floating-point instruction using a TOS addressing mode. To optimize performance, the compiler attempts to increase the length of the floating-point instruction sequences that can be pipelined.

Due to the relatively low performance gain and the implementation complexity of support for overlapping floating-point and integer instruction execution, the CPU implements only the mechanisms that support floating-point instruction pipelining. The CPU stops after issuing a floating-point instruction to the FPU if the next instruction is an integer instruction. The CPU starts executing the integer instruction only when all previously issued floating-point instructions successfully complete (the FIF becomes empty).

The decision to serialize the floating-point instruction execution when an integer instruction is encountered eliminates the need for CPU shadow registers. Because the CPU stops when it encounters floating-point instructions that either use the TOS addressing mode or update the PSR, the only parameter the CPU needs to save in the FIF is the PC of the floating-point instruction (Figure 10). It saves this PC to recover in case of a floating-point exception. The CPU saves the PC value in the FIF when it issues the corresponding floating-point instruction to the FPU. An SDN signal simply advances the FIF. As a result, the FIF output always contains the PC of the floating-point instruction currently ready in the FPU.

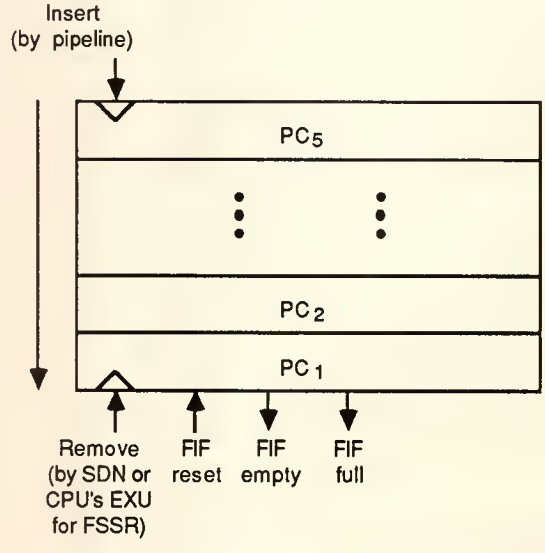


Figure 10. FIF configuration that supports floating-point instruction-execution pipelining only.

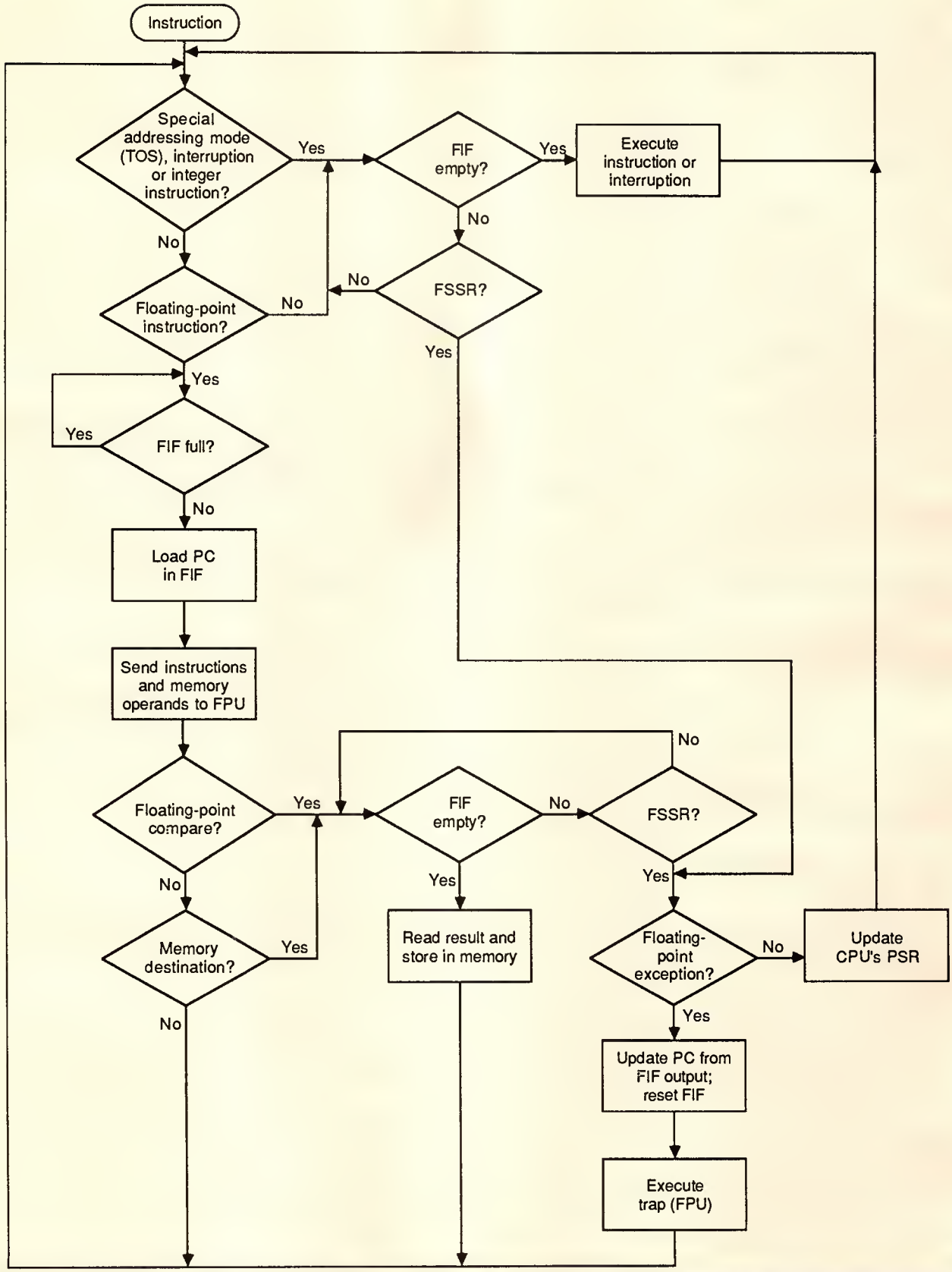


Figure 11. CPU instruction execution for a pipelined slave interface.

The instruction flow through the floating-point, instruction-execution pipeline is the same as that shown in Figure 4. The flowchart in Figure 11 details CPU support for pipelined, floating-point instruction execution using the simple recovery mechanism previously described.

When the next instruction to be executed is an integer instruction, the CPU's EXU serializes the instruction execution. That is, the new instruction does not start executing until all the pending instructions finish successfully.

Otherwise the CPU saves the instruction's PC in the FIF and issues the instruction to the FPU for execution. If the issued instruction is a floating-point instruction with memory destination or if it is a FLOATING-POINT COMPARE instruction, the CPU's EXU stalls until all pending instructions finish. Then the EXU can continue, storing the result to memory or updating the PSR, respectively. Otherwise, the CPU's EXU can continue immediately with the next instruction.

If a nonmaskable floating-point exception occurs during the execution of floating-point instructions, the CPU restores the original system state by replacing the current PC with the PC value at the FIF's output and flushes the FIF. Only then will the CPU trap with the Trap(FPU) vector. This simple mechanism maintains the precision of 32532 floating-point exceptions.

The FPU's input FIFO, which saves the incoming floating-point instructions, has the same depth as the CPU's FIF. At the same time, the FPU asserts signals indicating the outcome of the floating-point instruction execution (SDN or FSSR) in the same order in which the CPU issued the corresponding floating-point instructions. As a result of these two conditions, the CPU stalls when the FIF becomes full (Figure 11) before the FPU's input FIFO overflows. Consequently, a handshake is unnecessary to signal the CPU that the FPU cannot accept more instructions.

For FPU chips using very fast data paths, the efficiency of the CPU-FPU interface becomes the critical factor for floating-point performance. A serial protocol that forces the CPU to wait for the outcome of the FPU floating-point instruction execution significantly degrades performance. The floating-point performance of the CPU-FPU cluster can be, in this case, several times lower than the performance of the FPU's data path.

The 32532 microprocessor features a pipelined slave protocol that hides the CPU-FPU communication overhead for most floating-point instructions by pipelining their execution. The FIF, a simple recovery mechanism implemented within the CPU, maintains the precision of floating-point exceptions. As a result, the 32532 microprocessor supports very high floating-point performance without sacrificing software compatibility with previous Series 32000 CPU-FPU clusters.

Acknowledgments

I thank ChakChung Ng, who generated the instruction distribution in Figure 9, and Don Alpert, Shay Ben-Chorin, Danny Biran, Jonathan Levy, Benny Maytal, David Schanin, Yom-tov Sidi, and Ran Talmudi, who reviewed the pipelined slave interface concepts and provided very useful feedback.

References

1. "WTL3164 Preliminary Technical Information, Rev. 0.2," Weitek Corp., Sunnyvale, Calif., Apr. 15, 1987.
2. J.E. Smith and A.R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," *Proc. 12th Ann. Int'l Symp. on Computer Arch.*, Boston, June 1985, pp. 36-44.
3. *Series 32000 Instruction Set Reference Manual*, National Semiconductor Corp., Santa Clara, Calif., June 1984.
4. J.T. Coonen, "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic," *Computer*, Jan. 1980, pp. 68-79.
5. *NS32532 High-Performance 32-Bit Microprocessor*, National Semiconductor Corp., Oct. 1987.

Sorin Iacobovici is a senior computer and system architect with National Semiconductor in Santa Clara, California. His experience and interests include computer architecture, high-performance computer systems design, and performance analysis and modeling.

Iacobovici holds an MSEE degree from the Polytechnic Institute of Bucharest, Romania.

Questions concerning this article can be addressed to the author at National Semiconductor, M/S D3678, 2900 Semiconductor Drive, Santa Clara, CA 95051.

Department

MicroLaw

Richard H. Stern
Law Offices of Richard H. Stern
2101 L Street NW, Suite 800
Washington, DC 20037

Protection against piracy

The following letter prompted responses from MicroLaw editor Stern and from Denis Karjala, Arizona State University professor of law, who was quoted in last issue's column. We invite your comments as well.—Ed.

To the Editor:

Richard Stern has the *wrong* idea ("The Berne Convention: A Bad Idea Whose Time Has Come?" *IEEE Micro*, Apr. 1988, p. 6).

Berne adherence is a *good* idea whose time has come. Why? Because of what's changed in the 100 years that Stern says we've "gotten along without" Berne: technology.

US authors and copyright proprietors now lose billions of dollars annually from piracy abroad. Computer programs are particularly susceptible. They can easily be copied outright and not only sold within the pirate country but also exported to many other countries throughout the world.

US adherence to Berne will make US authors and copyright proprietors better able to enforce their rights against foreign pirates. Until we adhere to Berne, US authors can achieve protection under Berne only through expensive and uncertain "simultaneous publication" in a Berne country. IBM spends millions of dollars each year on simultaneous publication, but for many individual authors and small software producers, the cost is prohibitive, and they must forgo Berne protection.

Because Berne doesn't have any effect on how the US protects US authors, our joining Berne won't change what Stern is apparently concerned about: the scope of US copyright protection that Congress and the courts have established for software. What it will do is enable the US to better fight something that Stern is apparently not concerned about: international piracy.

Gunter Hauptman
Purchase, New York

Stern replies:

The views of Gunter Hauptman, one of IBM's most senior experts on intellectual property law, always deserve careful consideration. But it is unclear from his very brief comments here how adherence to the Berne Convention would really help US software writers, users, and sellers more than it would cost them—or, more generally, how Berne would advance software progress in the US more than it would hinder it. I hope, therefore, that he will agree soon to give *IEEE Micro* readers a more detailed analysis of the "pros" and "cons" of Berne, as seen from the unique vantage point of his considerable international experience with the major international factor in the US electronics industry.

Such an analysis would address questions such as these:

Why is Berne protection necessary, in addition to Universal Copyright Convention protection (long available to US authors and publishers)? All of the major software markets in which a small US seller would be interested seem to have adhered to both of these treaties, making Berne protection superfluous for these Americans. Is there a real need? For whom?

How would our joining Berne leave us free to drop "look and feel" protection from our copyright system, or to substitute short-term non-Berne protection for it—without violating our Berne treaty obligations? The same question may be raised for other noncode aspects of computer programs, such as instruction sets, languages, and selections of algorithms to use.

I understand that the last of these was an issue in the NCR/Comten/IBM litigation, and it remains an issue of concern whether our present copyright law covers that aspect and whether it ought to. If Berne will not leave us free to be flexible about whether and how to protect noncode aspects of computer programs, such as selections of algorithms, can we

afford to buy into Berne? This is one of the major questions not addressed in detail in the Hauptman letter, and one to which *Micro*'s readers would greatly benefit from further explanations.

Why would it be all right under Berne, as the letter indicates, to give US authors less protection than foreign authors?

If Berne allows that, why would we want to have a set of copyright laws giving foreign sellers of software greater rights than US sellers? What would the economic consequences be? Will we end up with people in the software programmer, seller, or user community saying, "Here's DRAMgate again?" I would like to hear more details about a proposal that we should have a system in which US owners of software copyrights get lesser rights than foreign owners. Maybe it would work out beautifully, but surely some of us would like to know more about the small print before signing on the dotted line.

How would Berne cause the US "to better fight...international piracy" of software? Why would Berne be any better at this than the present US treaty, the Universal Copyright Convention?

If Berne would accomplish this, we may have a reason to adopt Berne. But will the benefits here exceed the costs? The costs seem to include having to drop our system requiring registration, at least for foreign software sellers or copyright owners; being locked into long-term Berne-type protection for the benefit of at least foreign software sellers; and the other costs of Berne. Who gets the benefits and who pays the costs? How much is to be paid? Even assuming that the net social value of the benefits to the "getters" exceeds the cost to the "payers," do the "payers" want to accept this kind of transfer from their pockets to those of the "getters"? This choice can be like the DRAM problem again. (What is good for the chip maker is not always good for the box or board maker, and vice versa.)

Contrary to the final unsupported suggestion of my good friend, Hauptman, I am concerned about software piracy. But I cannot get so worked up about that concern that it overrides every other consideration. Everything has its limits. There are always other competing interests at stake.

For example, because I also have other concerns I would not advocate capital punishment or autos-da-fe for software pirates. I would also not like to sign on for 50 or 75 years of injunctions and jail terms against anyone's duplicating the </FR> keystroke pattern of I-2-3, or use of high-intensity capital letters to designate the keystrokes for a command in a menu (for example, **QU**it in Crosstalk), or selections of which algorithms to use in a program. If the Berne controversy forces me to choose between my concern against international piracy and the latter concerns, I will reluctantly choose the latter concerns as more pressing or higher on my agenda. Others may share that view.

I would like to see a fuller explanation of why we need Berne; what it will cost software programmers, users, and sellers; and how all of the relevant trade-offs will balance out. Gunter Hauptman's short letter is a step in this direction, but it is too terse to do justice to the merits of the pro-Berne position. I do not want to buy a pig in a poke, even one that comes tied up with a nice blue ribbon. That is why I hope to see in *Micro* an expanded version of the Hauptman letter that addresses all of these issues in detail. Only then can we judge whether Berne is a good idea or bad idea whose time has come.

Karjala's comments:

Gunter Hauptman's statement that US copyright proprietors "now lose billions of dollars annually" from foreign piracy is almost surely overstated. But even if correct, the claim is only marginally relevant to whether the United States should join Berne. Every economically advanced country that is a member of Berne is also a member of the Universal Copyright Convention (UCC), under which works by US proprietors are protected. As of January 1, 1986, some 22 countries were members of Berne but not the UCC (derived from tables in *Nimmer on Copyright*, M. Nimmer, Vol. 4, Apps. 21, 22, 1987). While Egypt, South Africa, Thailand, and Turkey are among the 22, so are Chad, Libya, Togo, and Zaire. Annual copyright losses of billions of dollars in those countries are implausible to the point of incredibility. In short, while piracy is indeed a problem with software technology, both at home and abroad, the question is what effect joining Berne would have on the problem.

Hauptman is correct that our joining Berne would not in itself affect how we protect United States authors here at home (at least those who first publish their works here), but it is unrealistic to think that we would allow greater protection to foreign authors than to our own. (And if we did, US authors who could afford to do so would make every effort to publish first in a different Berne country.) The Berne standards would then apply here for such major players, leaving only the "individual authors and small software producers" for whom Hauptman has so much concern without the supposedly higher standards of Berne at home, where they most desire them.

Once we join Berne, it will be much more difficult, both politically and otherwise, to experiment with compulsory licensing systems and shorter periods of protection for software. Thus, a system of intellectual property protection designed for works of art, literature, and music, and applied only with difficulty and inconsistency to telephone books and other useful articles containing expressive human communication, becomes frozen in application to a rapidly developing new technology for making computing machines perform work.

So far, copyright law has proceeded almost blindly in protecting software, because nothing in its venerable, if somewhat incoherent, history has prepared it for this new task. We should not cast the initial tentative steps in stone before the basic policy issues and their resolution are in clearer focus. (For my extended analysis of those issues, see "Copyright, Computer Software and the New Protectionism," *Jurimetrics J. of Law, Science and Technology*, Fall 1987.)

I agree with Gunter Hauptman on one point: The underlying motive for activating the movement toward Berne now is technology protection, in particular, computer software. But if protection of technology is the issue, why have the congressional hearings been concerned nearly exclusively with a parade of witnesses discussing moral rights and copyright formalities? Missing from the debate is the public interest in the free flow of technological information and methodology. It seems that whoever or whatever is orchestrating the Berne steamroller has found it useful to divert attention from the real issue.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 171 **Medium** 172 **High** 173

Department

MicroReview

Richard Mateosian
2919 Forest Avenue
Berkeley, CA 94705-1310
(415) 540-7745

Pocket Modem (Migent, Inc.)

Migent has made a tiny Hayes-compatible modem that fits in a shirt pocket and runs on a nine-volt battery. The people at Migent sent me the Macintosh version, which simply means that it was shipped with Macintosh communications software (Borland's MacTerm 2.0) and a cable for a Macintosh Plus, Macintosh SE, or Macintosh II. However, when I tested it, I used a Compaq 286 Portable.

One of my clients has a 3-Com network, and the remote call-in software for that network is considered a stringent test of Hayes compatibility. In fact, I was told that I'd better not even bother to try to call in unless I had a genuine Hayes modem. So when the Migent Pocket Modem arrived, I removed it from the package, inserted a battery, connected it to the phone line and to the Compaq, and started the communications software. It connected and ran without a hitch, so I'll pronounce it thoroughly Hayes compatible.

(By the way, the 3-Com communications software may be good for testing Hayes compatibility, but it's not much good for anything else. It moves far too much data during simple operations for it to be considered usable at 1200 baud. I'm not reviewing the 3-Com software; I'm just making an observation.)

The Migent Pocket Modem is brightred. It is 2.5 in. wide and 5 in. long. At one end is an RS-232 25-pin connector, and from that end for about half its length the modem is less than 0.75 in. thick. Then a battery compartment widens it to 1.25 in. The modem fits into my shirt pocket, protruding upward less than half an inch, but it is a little bulky to be carried there comfortably.

At the opposite end of the modem from the RS-232 connector are two telephone connectors and a small round socket for connecting an AC adapter to allow operation without a battery. One of the connectors connects to the phone line, and the other can be used to attach to a telephone to allow monitoring of the progress of the call (the modem has no speaker of its own).

The Pocket Modem has no lights or switches. It emulates Hayes switch settings by nonvolatile memory, which must be configured using the same protocols as are used with the Hayes to override switch settings. Migent ships a setup program with the modem, but it is not required when using the Borland software. The modem is shipped with settings that correspond to the factory settings of the Hayes modem switches. Since it has no power switch, the Pocket Modem draws current from the battery whenever the computer it is connected to is running. Depletion of the battery can be avoided by using the AC adapter. The Pocket Modem comes with battery, 7-foot phone cord, AC adapter, computer cable, a brief but excellent manual, and some basic software.

At this point I have to say that my other modem is a Hayes Smartmodem 1200, from which I have had trouble-free operation for the approximately four years that I've had it. I don't know what Hayes modems sell for these days, but I'm sure it's several times the price of the Pocket Modem. Migent has found a way to package the essential elements of the Hayes into an attractive product, but I'll never be tempted to disconnect my Hayes and replace it with the Migent. The Migent is great for traveling, but for my home base I'm happy to pay for those Hayes "inessential" elements, like a speaker, lights, power switch, and a history of quality and reliability.

Reader Service Number 2

Draw It Again, Sam... (Aba Software, Inc.)

This is a frustrating product. I liked the people I talked with about it at MacWorld Expo in January. I like the idea of the program and the intelligent way it's been designed. I like the organization of the manual and the way it proceeds slowly and systematically from the simple to the complex. Yet every time I look closely at this program

or its manual, I see evidence of carelessness.

The first example of carelessness I noticed appeared early in the manual, where a great point is made about using up-to-date System, Finder, and device drivers. The manual lists the required versions of these files, including LaserWriter and ImageWriter drivers, followed by the statement that "the System, Finder, and driver files provided with this release of the program are sufficient for using the product." Since some of the required driver versions were more up to date than those in my system folder, I decided to use the ones provided. Unfortunately, there were no driver files provided, nor was there room for them on the shipped disk.

This example of a mismatch between the product and the manual accompanies plenty of examples of errors that are entirely confined to the manual. Gross typos abound, and the English is often clumsy. Some of it looks as though a rough draft were rushed into print without anyone, even the author, ever looking at it.

Another example of carelessness occurred the first time I tried the program. I opened one of the sample drawings and used the Zoom Out option to get an overall view. Careful reading of the manual tells you that you can only zoom out five times, giving a 32-times reduction. What the manual doesn't tell you is that if you invoke the Zoom Out option a sixth time, the program goes into never-never land, and you have to reboot.

I won't go on with this carping, but I'm not sure I'd want to have to depend on this product for anything. That comment aside, however, the program has been designed well and implemented nicely at the gross level (the carelessness seems to creep in where details are involved). The basic idea is to combine an object-oriented drawing program with a bitmapped painting program. Drawing can occur on different "layers" or planes, which are isolated from one another, except when special commands move objects between layers or select groups consisting of objects from more than one layer. While you are working in one layer, objects in another layer can be visible, invisible, or "grayed." Graying seems to be a compromise between the efficiency of "invisible" and the convenience of "visible." Apparently screen image regeneration becomes time consuming when drawing operations are performed on complex drawings.

One of the main uses of layers is in the preparation of color separations. In addition to providing for the duplication of objects in multiple layers and the separate coloring of those layers, the program also provides the registration marks that printers use in combining color separations into the final printed drawing.

The program implements all of the familiar drawing and painting features of the MacPaint and MacDraw programs and a number of others as well. Switching between the drawing and painting tool palettes is extremely smooth, and the use of menus and submenus is also good.

As I said earlier, I really like this program and its manual, so I find glitches of the type I've mentioned extremely frustrating.

Reader Service Number 3

***Microcomputer Hardware Design*, D.A. Protopapas (Prentice Hall, Englewood Cliffs, N. J., 510 pp.; $44)**

In August 1987 I reviewed Michael Slater's *Microprocessor-Based Design,* a book I had many good things to say about. Since then I've received a number of books on microprocessor interfacing. This is the best I've seen since Slater's.

Protopapas starts with a generic introduction to microprocessors and microprocessor-based computers, followed by a discussion, still at a fairly general level, of the Motorola 68000 and Intel 8086 families of microprocessors. Chapters on memory, input/output, and peripherals complete the book. The book gives no indication that the author has used the material for a course, but he has organized it as a textbook aimed at college seniors or graduate students in computer science. Each chapter ends with an extensive set of problems. Examples stated in the form of solved problems appear throughout the book and form the basis for understanding how to solve the problems at the ends of the chapters. A solution manual is said to be available to instructors, but I haven't seen it, so I can't comment on its adequacy. The solutions in such manuals tend to vary considerably in quality.

The material covered by this book represents a good selection and is presented in sufficient depth to be useful. While Protopapas surely knows this material well, his writing does not convey the same feeling of intense involvement that I felt from reading some chapters of Slater's book. I conjecture that this difference in feeling stems from the difference between academic and industrial experience with the material. My own background, which is more industrial than academic, makes me prefer Slater's book, but someone else might reasonably prefer Protopapas' book, especially for use as the principal textbook for a course.

***Silicon Compilation,* Daniel D. Gajski, ed., Addison-Wesley (Reading, Mass., 450 pp.; $43.25)**

In his preface, the author explains the approach he has taken to the material. Other books do now and will in the future take the standard textbook approach, developing generic material systematically. He has instead assembled chapters on a variety of actual silicon compilers. Following some introductory chapters, he presents descriptions of eight silicon compilers, each written by representatives of the firm or institution that developed the compiler.

This approach gives the book the flavor of an extended trade show session. This is not a negative criticism, however, since I know that many engineers flock eagerly to shows like Wescon and Electro every year to obtain information presented in just this format and not available easily from any other source.

The big problem with trade show sessions is that the material presented and published is not reviewed in advance. Material varies greatly in quality of content and appearance, and it is often biased and self-serving. This book, while potentially subject to many of the same problems, seems to have avoided them to a large degree. Attractively produced, the book presents the text of all chapters set in a uniform typeface. I did not notice anything that looked more like a marketing pitch than a technical exposition.

For anyone interested in silicon compilation, this book is well worth reading.

MicroView

HyperCard screen. (Courtesy of Apple Computer, Inc.)

HyperCard: Tool or toy?

Christine Miller, Assistant Editor

Apple Computer's HyperCard has attracted a great deal of notice because of the unique way it allows users to organize information. Users may associate or link data any way they wish—from current production figures to next week's office appointments. HyperCard's creator Bill Atkinson calls it a software erector set, Apple calls it an authoring tool, and now universities are beginning to use it as an audio visual aid *extraordinare.*

In the educational context, HyperCard functions like an electronic reference librarian: Students associate bits or volumes of information on the screen while the subject is still fresh in their minds—without running out of fingers to mark their places in the textbook. Learners participate in coursework by computer (we've all seen grades go up on this account) and even add information to the courseware databank (under supervision).

Inquisitive about how a science professor might develop HyperCard courseware, *IEEE Micro* interviewed Bob Gross of the department of biological sciences at Dartmouth College. Gross holds a PhD from Johns Hopkins University in biophysics, has been selected for inclusion in *Who's Who in Frontier Science and Technology,* was a panel member of the Annenberg/CPB project for funding PBS educational projects, and has published 11 articles in his field. His work at Dartmouth now serves as an example in Apple's courseware development program.

We've heard a lot about HyperCard's capability to organize one's office activities, but you've put HyperCard to work in quite another application—teaching science. What made you think of using it this way?

I view HyperCard not just as a data manager, but as a way of managing your thoughts, making connections. It allows users to make connections between pieces of data. I can connect one thought to another by using just two steps to create a button in HyperCard.

A HyperCard stack can be used as a teaching tool or in research—we have it running on our computer in the lab. I have been developing some teaching stackware called "Techniques in Molecular Genetics." Say a student wants to know more about the term "centrifugation." He just presses the button over that term on the screen and is taken to a stack of cards that deal with and animate centrifugation. On every single card in the stack there's a place to add notes. And you don't lose your place. [*Each Hypercard stack comprises a number of user-created cards linked by buttons or use of the HyperTalk programming language.—Ed.*]

The second part of the coursework is the question stack. I plan to have students write five questions and answers on the techniques of molecular genetics. They'll have to do the research and understand things and hand in the stacks they've created. It will enable them to learn things more easily and probably have a lot more fun. I mean, it's learning right off the computer—it's not sitting down in the dorm with a can of soda and a chocolate bar, trying to memorize things.

Do you think HyperCard adds a formal discipline to learning?

No, I think it makes it more informal, really. It's not that structured. The ability to jump around in a nonlinear way allows you to access all the information on a given subject that may or may not be peripheral.

Also, I've created a number of movies that illustrate some of the techniques in the stack. I run these VideoWorks movies through HyperCard using a driver from MacroMind. [*VideoWorks II is a color animation-creation system.—Ed.*] This combination allows students to see things moving around on the screen. It's wonderfully exciting—they bring their friends.

Last week I tried using HyperCard for the first time, giving a 15-minute demonstration. Students seemed very impressed. During downtime from lab procedures, students who had never before used a computer sat down without any instructions and were writing questions and exploring things and watching movies.

A visual image of a procedure happening in front of you is a lot easier to understand than reading words.

What was your specific objective in using HyperCard?

I wanted to provide students with a set of accurate notes that they could explore on their own through an interface that eliminated boring, tedious study. I believe students spend more time on their work when they enjoy it.

What did you produce?

The main stack about techniques, the question stack, and an indexed glossary. They are all interconnected; you just don't realize it when you press the button. The glossary is a neat little stack that allows the student to define any word by clicking on it.

Did you use sound and graphics? How did you accomplish this?

VideoWorks animations are very sophisticated, with a lot of movement and some digitized sound as well. Within HyperCard itself you can also do some animation, but it's not as flexible as VideoWorks. There are some very nice effects in HyperCard, like dissolving one screen into another. In one of the techniques that we use, there is an enzyme capable of chewing up RNA. Using HyperCard, I can make an exact duplicate of a card containing an RNA component and then erase that component on the second card. Then I dissolve from the first to the second card, very slowly. The only thing that appears to change on the screen is that the RNA disappears.

Now, let's talk about the actual building of stacks. Danny Goodman [*author of* The Complete HyperCard Handbook *and Focal Point software*] has been quoted as saying that users can build simple stacks without knowing a word of HyperTalk programming language. Was that your experience?

Yes, absolutely. You build simple stacks and make the buttons go from one place to another.

Did you get into something a little bit more complicated than that?

Yes, I wanted to be able to do some pretty sophisticated things, like accessing and playing VideoWorks movies. I also changed the fonts and some of the buttons, which required doing a little bit of radical scripting. But scripting wasn't really necessary.

Was scripting difficult for you, or was it easy to learn?

Well, I've done some programming. I'm very familiar with Basic and I know a little Pascal, so I didn't hesitate at all in going into HyperTalk language. The language itself is very easy to learn. It's a very nice, structured environment. You really do object-oriented programming: code associated with a button, code associated with a text field, code associated with a card. And they all interact and pass messages back and forth. Apple never uses the word "programming" in relationship to HyperTalk. I suppose they don't want to intimidate the user, who would probably say, "Oh, no, if I want to use HyperCard, I'll have to learn how to do object-oriented programming." They want to avoid that.

What equipment did you use?

To do it efficiently, you need a hard disk. One of the most useful parts of HyperCard is its help stack, which is 737 kilobytes in size. The help stack is a tutorial and a reference guide all built into one. It's a set of HyperCard information in a hypermedia [*interactive multimedia*] format, so you can jump around and get whatever information you want very easily. It could be done with two floppies. It just happened that I had a Mac Plus in my office with 2 megabytes; I have a Mac II at home that I used for the same kind of development.

"A visual image of a procedure happening in front of you is a lot easier to understand than reading words."

What was the cost?

I didn't get paid for the time that I spent. I would ordinarily make slides or overheads for a course—I just used HyperCard instead. The card came with the Mac that I bought, so it was essentially free. The VideoWorks I came with a course budget several years ago. I happened to get to know Marc Canter of the MacroMind company, and he gave me a free upgrade of VideoWorks II. I think you can buy II through a mail order place for $120 or $130, I'm not sure of the price. Of course, the materials were negligible—a $1.12 disk.

How long did it take? How many people? With what background and training?

I've been working at it on and off—not steadily—since last July. Making a movie might take a day or two; putting it in the stack takes about a minute.

A couple of students helped me to make movies, and now Dartmouth has engaged an art major—who intends to go into computer graphics art—to design the new movies and graphics. The stack itself, the format, and the connections are all mine.

What were the challenges in developing your stackware?

Technically, there were very few. They mostly had to do with the interface. I expect this will be solved in the next HyperCard version that comes out. I really don't want to go into it, but it was very difficult to do a good job in putting together the information in a way that is easy and intuitive to follow. It's very easy to throw stackware together in a haphazard way and get it to function, but then people wouldn't be interested in using it. It takes a lot of time and experimentation. That's the biggest challenge: to make it intuitive, consistent, useful, and fun.

What other problems do you expect the next version of HyperCard to solve?

Because of my involvement with beta testing, I can't discuss the limitations of HyperCard, but you can check with Compuserve and users groups for the limitations. Apple itself has said it will keep developing HyperCard for the next three years. My guess is that after two or three years, Apple will make a ROM version of HyperCard and stick it in computers so that it will become a part of the operating system. That will be pretty impressive. Bill Atkinson wrote HyperCard—a revolutionary program that is really different from anything else out there. If Bill Atkinson is going to spend two years on it, which he's doing, it's going to evolve into something pretty nifty.

What should the first-time stack author keep in mind?

The most important thing in stack design is to consider the person who's using it. They're coming into it absolutely cold. What will they perceive? You should keep this in mind as you design and evaluate the outcome. Also, start with something fairly simple—add to it. Try to plan ahead. And lastly, explore! Don't be afraid to try things.

Reader Service Number 4

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 186 **Medium** 187 **High** 188

Department

MicroNews

MicroNews features information of interest to professionals in the microcomputer/microprocessor industry. Send information for inclusion in MicroNews one month before cover date to Managing Editor, IEEE Micro, *10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.*

Technology research: Ultrafast devices, X-ray lithography, and micromachines

What's new in research? Here's a quick look at some of the latest projects in gallium arsenide devices, digital and analog circuits, X-ray lithography, and micromachines.

GaAs device

Stanford University's Ultrafast Electronics Laboratory researches simple chips in hopes of pushing the frontiers of speed. Engineers chose GaAs to design and build an IC with signals corresponding to 100 GHz. The researchers say the IC is so fast that it pushes their own abilities to measure it.

The 2-inch-wide GaAs wafer clocks in at around 3.5 picoseconds when operating at room temperature.

Although the major application for the new IC will be to make electrical measuring instruments with better time resolution, researchers hope it will lead to even more complex systems capable of working at the ultrafast speeds.

Digital circuit

Other Stanford engineers in the Center for Integrated Systems hope to develop a digital adder circuit that adds two 32-bit numbers in 1 ns, a feat that impressively improves on the currently accepted 10-ns time for the same operation.

Professor Michael Flynn and doctoral candidate Gary Bewick incorporated three approaches to addition in their circuit design: conditional sum algorithm, carry-lookahead, and a modification of the way the adder circuit defines carries. Their circuit performs 32-bit addition with three gate delays, a reduction which makes the circuit extremely fast.

Though the speed of light limits circuit speed, Flynn and Bewick plan to reach the highest speed they can by making the device smaller. "We'll fabricate a significant part of the machine on a single piece of silicon, and this (adder) will be part of that piece of silicon," Bewick says. "The more we can fabricate on a chip, the faster the machine will go."

Flynn and Bewick hope to receive the first fabricated circuit from Signetics Corp. in Mountain View, California, soon—and proceed to testing. Signetics agreed to fabricate the prototype after associate professor Giovanni DiMicheli created a computer program that allowed the Stanford design to work with the company's fabrication process.

"Our longer term goal is to see if we can bring all of the arithmetic operations, or as many as possible, into a rate of at least one per nanosecond," Flynn states. "We have other techniques we're looking at, mainly concerning the inherent delay in circuits and using that delay to our advantage."

Analog circuit

A University of Southern California researcher chose to rely on past technology to pursue the electronic dream of virtually instantaneous information exchange. Associate professor John Choma, Jr., redesigned an old-fashioned analog circuit to work in the ultrafast signal-processing future. He has reached 6-GHz performance in some silicon test cells he's worked with so far.

"Electrical engineers always knew the old analog circuits had a faster processing capability than today's digitals," says Choma. Digitals can process signals up to 400 million times a second, but analog circuits fabricated in silicon theoretically could process signals up to about 7 billion times a second.

Despite this speed advantage, analog research stopped around 1970 because digital circuits proved more economical, took up less space on a chip, used less energy, and were easier to manufacture. Now that the speed limits of computers and other systems are being challenged, digitals may have to move over so analogs can have another chance to overcome the speed problem.

Processing the analogs is so sensitive that "even a hiccup can affect the obtained speeds," Choma reports. Of the 15 test cells so far produced, only three or four have worked, largely due to the lack of available, state-of-the-art processing sites and reliance on less-developed but within-budget sites. Choma's project, it seems, remains on the low-priority list of projects at the better-equipped sites.

Though the new design takes more chip area, the loss of space for transistors is no problem. "With analog circuits, you generally need only a small fraction of the number of transistors now needed with digital circuits," reminds the electrical engineering/electrophysics professor.

To fight the problem of reducing the extra manufacturing cost inherent in maintaining the signal quality, the researcher approaches industry representatives. He recently visited General Electric, Valley Forge, which is interested in analogs in connection with its plans to process cells for satellite data. Other interested companies include TRW, Silicon Systems, Semiconductor Research, Polycorp, and Westinghouse.

"Meanwhile," he reports, "I'm working on the possibility of designing analog

circuits to work still faster. I suspect we may have just scratched the surface, as far as analog circuit speed is concerned." Research into GaAs analogs comes next term with possible investigation into high-electron mobility designs.

Choma realizes analog circuits will not replace digital circuits in every application. "But," he states, "analogs definitely have a part to play in speed-important spacecraft and military applications."

X-ray lithography

Computer chips may continue to shrink in size, if support for the comparatively new X-ray lithography technology continues to receive industry and government support, say representatives from the University of Wisconsin-Madison.

X-ray lithography technology permits small and dense circuit patterns to be etched on silicon chips, leading to reduction in the end size of the chips.

To date, UW-Madison researchers have worked on several lithography aspects, including that of developing the photosensitive chemicals known as resists, which are used to coat the silicon wafers. Programs for fabricating the photographic negative-like masks and for computer modeling the process also contribute to the university's dominant research position.

But the most dramatic of the programs has to be the university's Synchrotron Radiation Center in Stoughton. SRC contains a large electron storage ring called Aladdin. The $15-million National Science Foundation-supported ring, one of only three in the country, produces the soft X rays needed to etch the circuit patterns. Five of the ring's 36 ports or beamlines support lithography research.

Aladdin is one of the two storage rings in the country that has space available for industrial X-ray lithography. The other, Brookhaven National Laboratory's synchrotron port, is reserved exclusively for the use of IBM Corporation.

Congress, acknowledging the technology's potential, has recently stepped up funding for UW-Madison's research to $25 million. To draw increased industry support, the university plans to construct a $1-million support center adjacent to the SRC. It will contain 4,000 square feet of clean rooms, wet labs, and fabrication/inspection facilities.

Micromachines

Another newly emerging industry stretches the imagination even more. Micromachines are mechanical devices shrunken to microscopic dimensions. They are so small that 60,000 gears, turbines, and motors could fit in one square inch. When combined with computer circuitry and miniature input sensors, they could serve as microsurgical tools or robots connecting microelectronic circuits on chips.

AT&T's Bell Laboratories engineers have used silicon chip-processing techniques to make air-driven turbines and working gears with teeth less than one fifth the thickness of a human hair. University of California, Berkeley scientists have built movable mechanical parts, such as cranks, gears, springs, and joints. Other engineers have made miniature sensors to detect pressure, chemical vapors, and changes in speed.

This "real revolution in the mechanical world" still has major problems to overcome. For example, micromotors and mechanical elements still need to be combined with computer circuitry and miniature sensors to become complete systems. However, the useful micromachines may prove to be inexpensive because they can in principle be mass-produced on a chip using silicon technology.

Superbus meeting

The Computer Society's Superbus Study Group meets in Geneva this summer to continue work on the proposed standard. David Gustavson, coordinator of the Physical Layer Task Group, will update members on the status of the standard July 15, 1988, at CERN.

Superbus, now in the very early stages of development, has minimum requirements: a 1G-byte/s transfer rate; support for distributed caches forming a coherent shared-memory image; support for bus repeaters, both for concurrency and for interfacing to other buses; and a media-independent, board-edge specification supporting passive or active backplanes, or switches (crossbar), which allows various cost/performance trade-offs.

Gustavson reports these goals appear feasible in the immediate future by abandoning the traditional handshake signaling mechanisms in favor of source-clocked, split-cycle operation. One promising implementation uses differential ECL (emitter-coupled logic) signaling on a parallel-ring structure.

Persons interested in attending this meeting, or wishing to join the study group, should contact Phil Ponting at CERN, 1211 Geneva 23, Switzerland; or telephone (22) 83 27 01 for details.

MicroTidbits

Though AEC professionals using 2D CAD currently outnumber those using 3D CAD (49 percent to 31 percent), a **Ralph Head & Affiliates, Ltd.** survey found that the two forms of CAD ran neck and neck for projected purchases. The most sought-after AEC software products proved to be CAD applications tied with project management and cost estimating.

Dial (303) 494-4774 to access an automated computer clock-setting service (ACTS) offered by the **National Bureau of Standards**, now in a six-month test phase. Time can be checked with 1/10- to 1/1000-second accuracy, depending on the mode of operation.

The NBS with seven other organizations signed a charter forming the **North American ISDN Users Forum**. The forum promotes the development and implementation of standard, interoperable products for the Integrated Services Digital Network. Join the forum by contacting Shukri Wakid at (301) 975-2904.

City and state governments approve desktop publishing purchases over those for minicomputer/mainframe systems by more than 16 to 1, reports **BidNet**, an electronic bidding information service. The publishing programs support community newsletters and the nearly 195,000 pieces of legislation introduced every two years in the state legislative process.

Commodore Amiga computer owners who use one Aegis Development Inc. product may enter the second annual **Desktop Video Contest** running through September 1st. Amateur and professional categories will be judged for best animation, special effects, computer and software use, artwork, creativity, editing, story line, sound, and overall quality and ingenuity. Direct any questions to Aegis at (213) 392-6445.

COS releases two test systems

The Corporation for Open Systems released two Open Systems Interconnection conformance test systems for vendors. The Message Handling Systems test includes test cases, licensing terms, international support capability and COS Mark Program compatibility. The File Transfer, Access, and Management test checks conformance with international standards.

The federal government and other large users have announced plans to require OSI conformance testing in their procurement of information technology products.

Interested vendors can purchase available conformance test systems by writing COS, 1750 Old Meadow Road, Suite 400, McLean, VA; or calling (703) 883-2756.

ASIC 88, Paris, France

James J. Farrell III

Societe Generale (a big French bank) had a rude surprise for me when I arrived at Charles de Gaulle airport this spring. I turned in $100 in cash and received back only 539 francs. The last time I performed this transaction—about two years ago—I received over 700 francs. However, no matter how disappointed and forlorn I appeared, 539 is all I got.

I paid 36 francs to a friendly Air France representative for a nonstop bus ride to the Palais de Congres, near the Arc de Triomphe in downtown Paris. The Palais is well known to European conference attendees as a really world-class facility for both exhibits and technical papers. This six-level facility has some very nice boutiques on the lower levels, large auditoriums for the presentation of papers in the mid-levels, which are surrounded by "exhibit hallways," and meeting rooms with typically excellent restaurants nearby on the top level. A large hotel occupies one end of the facility.

I ascended the familiar escalator to the fourth level to attend ASIC 88. Though the French registration form intimidated me a little, my limited written vocabulary was good enough to secure my attendee's badge.

For an initial effort in a fairly new electronic specialty like ASICs (application-specific integrated circuits), ASIC 88 did quite well. I would estimate that the exhibit floor held 40 or 50 "standard"-size (the European equivalent of 10 ft. × 10 ft.) booths, with several booths two or three times as large. I do not know what the total attendance was for the four days of the exhibition, but I saw several hundred people on the exhibit floor each of the two days that I attended. The exhibitors appeared to be a microcosm of the international ASIC semiconductor industry. The European affiliates of most US and Japanese companies exhibited in addition to virtually all major European manufacturers.

ASIC 88 was not the typical "hit-and-run" exhibit floor that I have become used to in the US. Attendees did not rush from booth to booth, gathering as much literature as possible or endeavoring to get their credit-cardlike badges embossed unto flatness. Most booths had operating CAD systems up and running, and their people seemed prepared to answer real ASIC questions on line, and in detail, for the attendee. One booth even had a cafe-table sitting area for lengthy technical discussions. It was not unusual to see attendees drinking wine and beer on the show floor as they examined the systems.

All in all, I felt I'd attended a very technically competent exhibit. I saw virtually none of the hype, hyperbole, magicians, trinket giveaways, or frantic hustling sometimes seen on US exhibit floors.

Still, I wish I could speak better French.

Current literature

The Guide to Futurebus (including a copy of the specification) provides hardware engineers with a working knowledge of IEEE 896.1 bus standard. The 76-page document details key features of Futurebus and discusses data transfer, arbitration, bus driving, metastability, geographical addressing, cache coherence, and live insertion.

Futurebus Information Service, Unit 2, Rowan Close, St. Peters Park, Brackley, Northants NN13 5UP, United Kingdom; $30.

HyperAge, the Journal of Hyper-Thinking, debuted this spring, calling itself "part of the shift that our society is making from the Industrial Age to the Information Age." The bimonthly publication promises to present theoretical and practical aspects of hypermedia, with strong ties to Apple's HyperCard, HyperTalk, and stackware products.

HyperAge, 5793 Tyndall Avenue, Riverdale, NY 10471; (212) 601-2832; $19 annual subscription.

A series of six reference manuals on transputers and transputer technology provide definitive material for designers, programmers, and scientists. The *Transputer Reference Manual* describes the architecture, the Occam model, and engineering data for IMS T800, T414, T212, and M212 transputers.

Prentice Hall, Publicity Dept., Prentice Hall Building, Englewood Cliffs, NJ 07632; $34.95.

Analyzing all major connectivity topics may seem to be a large task, but a Massachusetts-based consulting firm claims it has done so. *Corporate Connectivity: Concepts, Technologies & Trends* discusses the major technologies and future trends impacting the full interconnection of a company's computing resources.

DeBoever & Associates, 179 Great Road, Suite 220, Acton, MA 01720; (617) 264-0155; $395.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 183 **Medium** 184 **High** 185

New Products

Marlin H. Mickle,
University of Pittsburgh

Take your PC to the field?

Smaller, faster, and *better* are key words in the computer industry; the goal of miniaturization has become an outstanding fact. Desktops, portables, and laptops abound; and now handheld computers offer additional features and facilities including the ability to get about as far away from the clean room as one could imagine.

One example is the 4.5-lb., 16-bit RHC-88 handheld computer, which incorporates various desktop features with MIL-STD-810D ruggedness. Suggested applications range from performing utility line maintenance to car diagnostics to electronic warfare support employing artificial intelligence.

Two pounds heavier than the typical handheld, the 9.4 × 6.4 × 2.6 in.-RHC-88 functions within −20°C to +65°C temperatures. The company recommends an optional Elastomer or Membrane keyboard for underwater applications.

The RHC-88's MS-DOS 3.2, 128K-byte ROM operating system supports most IBM-compatible software programs; some require modification. Users can develop new software in Ada, C, Basic, Pascal, or Fortran.

The Paravant handheld field computer features a 74-key alphanumeric keyboard spaced for gloved-finger operation in demanding environments.

An expandable 512K-byte RAM main memory jumps to 1M byte with an optional card. The RHC-88 provides an additional 192K-byte ROM disk for program storage. Alternate power supplies consist of a rechargeable NiCad battery pack, five alkaline C-cells, or transformer AC current.

The 5 × 2.75-in. liquid crystal display screen is backlit to overcome darkness or poor weather conditions. In text mode, the screen displays 16 lines by 42 characters; graphics mode provides a 256 × 128-pixel resolution.

Data transfers at 19.2K bits/s via an RS-232 communications port. Additional features include two internal expansion boards, internal radio and telephone modems, and a full bus interface inside the battery compartment. The color-coded keyboard contains 52 tactile-feedback keys that provide 74 characters/functions. **Paravant; $3995.**

Reader Service Number 10

No longer bent out of shape

Cadra-II 5.0 software for Adra 1000 and 3000 CADD systems adds a flat pattern development feature for sheet-metal applications based on aerospace and automotive specifications.

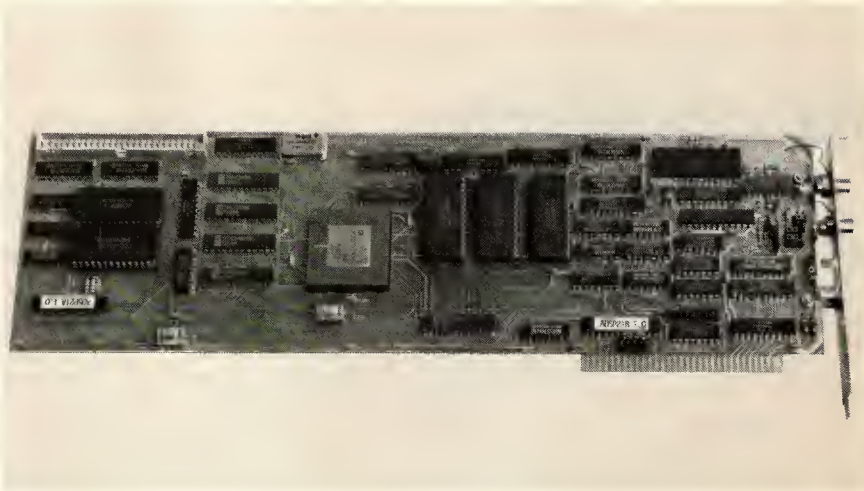
An Unbend menu function produces the flat pattern for bent sheet-metal parts where bend allowances are taken into account. A Flat Pattern menu function generates 2D blank patterns from the edge lines of bent sheet-metal parts. A drawing overlay function allows list association of 200 drawings to compare the fit between parts. Other 5.0 features include enhancements to the Auto-geometry CADD programming language, support for DIN-standard dimensions, upgrading of IGES translator to Version 4.0, and support for the Ethernet TCP/IP networking standard. **Adra.**

Reader Service Number 11

Develop your DSP

The 6-MHz or 8-MHz ADSP-2100 Development System offers MS-DOS computer users a development/applications environment for the 16-bit, fixed-point Analog Devices digital signal processor. The system includes a DSP plug-in board, debug monitor, Cross-Software development assembler/linker software, and simulation of the user's design architecture. On-chip arithmetic shadow registers allow context-switching completion in 125 ns. Concurrent with DSP processing, memory access transfers from the PC take two ADSP cycles. The debug monitor software supports breakpoints and register examination, while a linkable C library provides PC functions for integration of user applications. **Spectrum Signal Processing; $2995; $2595 without software and simulator.**

Full system **Reader Service Number 12**
Basic version **Reader Service Number 13**



The ADSP-2100 Development Board features 45-ns, 8K × 24-bit program memory and 8K × 16-bit data memory dynamically dual-ported to an IBM PC.

Portable, multilingual faxes

The 10-lb., portable F-20 facsimile machine contains a built-in telephone that stores up to 50 numbers. The F-20 attaches to the wall; a carrying case is optional.

The larger F-30 model lets users select English, Spanish, or French prompts (text translation is yet to come). Sixteen shades of gray scale aid halftone reproduction. With an optional B-4 printer, this fax produces both letter- and legal-sized documents without reduction. **Murata Business Systems; $1695 (F-20); $2495 (F-30).**

F-20 **Reader Service Number 14**
F-30 **Reader Service Number 15**

Desktop faxes

The Konimail 200 desktop facsimile transceiver transmits at a rate of 20 seconds per 8.5 × 11-in. page and features speed and delayed dialing, automatic polling, document feeding, and contrast control. The 11-second/page Konimail 400 offers 100-number memory with programming of automatic alternate numbers, confidential message reception, and automatic relay broadcasting. Both models communicate with CCITT Groups I, II, and III fax machines. **Konica Business Machines.**

Konimail 200 **Reader Service Number 16**
Konimail 400 **Reader Service Number 17**

What we need is more colorful plots

The CalComp ColorView 5612VS comprises a Model 5612 thermal transfer printer and a Model 903 color video controller.

The ColorView family of thermal transfer color devices outputs CAD/CAM/CAE color check plots and final plots in ANSI-B or ISO-A3 sizes. The 5612 printer produces a B-sized color page in two minutes and an A-sized color page in 75 seconds. Similar to the 5612, the desktop 5912 color plotter/printer uses an internal rasterizer to print 200-dpi hard copies. In addition, it emulates CalComp 5800 series electrostatic plotters.

The 903 color video controller supports both printers, connects to most analog RGB workstations, accepts 1280 × 1024-pixel-resolution images, provides 2M bytes of video memory, and allows full-screen resolution images to be produced independently of software. **CalComp; $13,995 (5612VS package); $9995 (5912); $7995 (5612); $8995 (903).**

5612VS **Reader Service Number 18**
5912 **Reader Service Number 19**
5612 **Reader Service Number 20**
903 **Reader Service Number 21**

Aural computing from Big Blue for the visually impaired

The IBM Personal System/2 Screen Reader allows blind or visually impaired users to hear screen-displayed text with the help of a text-to-speech synthesizer, aural documentation, and an 18-key keypad. Braille documentation is optional.

The keypad connects to the system's pointing device port, separating Screen Reader commands from keyboard functions. The user controls the pace and manner of spoken text—from character to full screen. Screen Reader's autospeak feature monitors up to 20 areas of the screen and alerts the user to error or status messages. Windows provide selection of blocks or columns to be read aloud. Compatible software includes IBM PC Local Area Network and Token Ring Network, Ashton-Tate's dBase III Plus, and Lotus 1-2-3.

More than 70 blind or visually impaired IBM employees tested the prototype. **IBM; $600 (without PS/2 configuration or text-to-speech synthesizer/related equipment).**

Reader Service Number 22

The IBM PS/2 Screen Reader enables the blind or visually impaired user to isolate a single column of text and hear it read aloud.

SCSI drive targets system builders

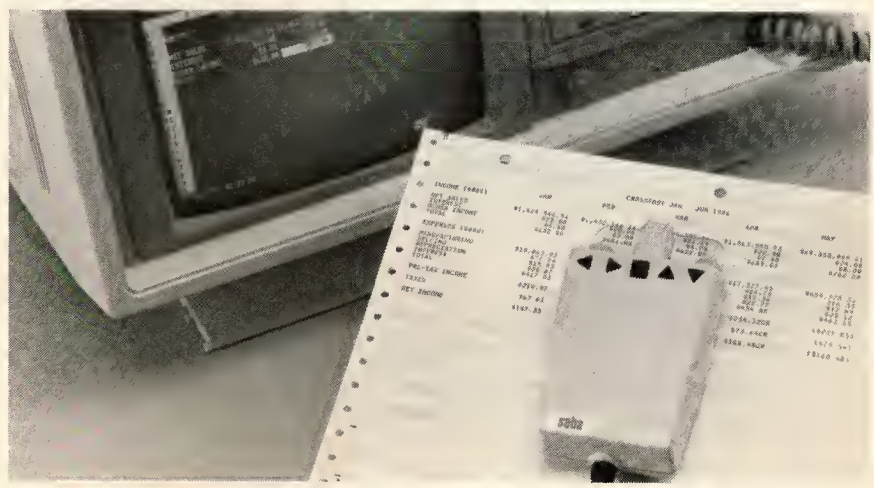
A Parallel Drive Array 1800 Series has been introduced with a Model 1804 SCSI drive, 1.5G-byte storage, and 4M-byte/s sustained data rate. According to the manufacturer, redundancy built into Model 1804 gives it a MTBF of 140,000 hours. The capacity and data rates are achieved with four synchronized, 38M-byte, 5.25-inch drives, each with 1.25M-byte/s data rate. When operating in parallel through an on-board controller, these component drives make the system appear to the host as a single drive with 5M-byte/s raw data rate. A fifth drive increases data integrity and system reliability. **Micropolis Corporation; $8/M byte (1000's).**

Reader Service Number 23

Scan that oddball

According to the company, its Apple- and IBM-compatible TurboScan/Flatbed scanner processes oddly shaped documents in 10 to 40 seconds without damaging them. Designed for desktop publishing and office automation applications, TurboScan/Flatbed scans bound materials, delicate or oddly shaped artwork, photographs, maps, reports, and letters. The scanner converts color or black-and-white text, handwriting, artwork, and photographs into 8.5 × 14-in. digitized images with up to 300-dpi resolution. Users can select line art and halftone modes or complete mixed-mode scanning in one pass. **AST Research; $1995 (IBM and compatibles); $1899 (Apple Macintosh).**

IBM compatibles
Reader Service Number 24
Apple Macintosh
Reader Service Number 25



Handscan Version 3.0 software allows IBM PC users to selectively scan portions of documents directly into spreadsheet, word processing, and database management software. Version 3.0 enhancements include reading documents from various line printers, expanded character substitution options, and support for the latest IBM keyboard. Saba Technologies; free upgrade; $799 others.

Reader Service Number 26

New Products

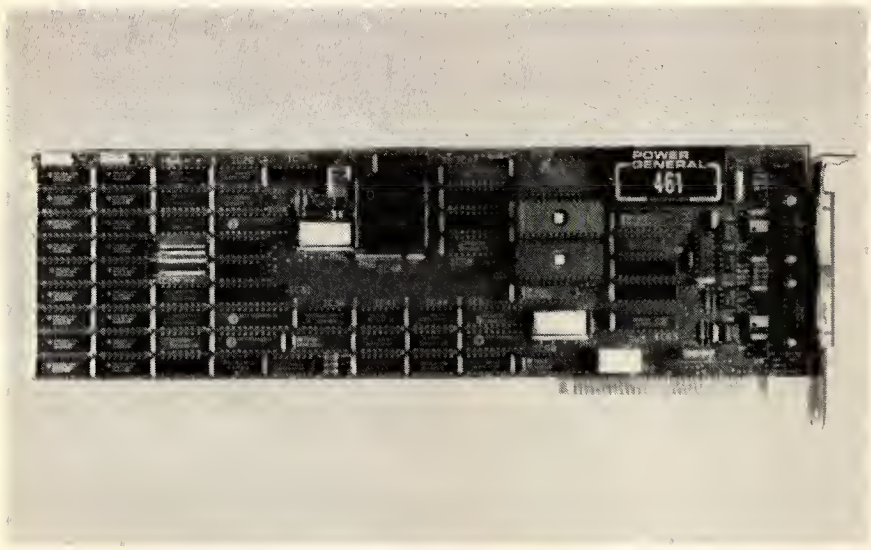
Amplifier has highs and lows, but few jitters

The AH20016 features low noise (70μV rms) and low feedthrough (–100 dB at 250 KHz) and holds an input signal to ± 0.0003 percent of full scale for 15 μs.

The AH20016 sample-and-hold amplifier was specifically designed for use with the AM40016 sampling A/D converter. With an acquisition time of 500 ns and 0.0015-percent linearity, the AH20016 offers a low-aperture uncertainty of 10 ps and a full-power bandwidth of 1 MHz. The amplifier can be used in front of a 16-bit A/D converter to digitize signals with frequency components as high as 485 KHz. Its low droop rate of 2 μV/μs enhances this use. **Analogic; $185 (100's).**

Reader Service Number 27

Real-time concurrent processor



Applications for the 7.4-MHz, 16-bit TM9000 include factory data collection, shop floor and inventory control, and production lot tracking.

The TM9000 plug-in transaction processor board adds 64 real-time, data-collection workstations to MS-DOS computers. Operating independently of its host, the 512K-byte RAM board provides prompting, editing, data buffering, and local validation.

Seven intelligent, multitasking TM9000's can connect to each PC. Because standard DOS device drivers accomplish the interface, multiple-board configurations do not need COM1-COM2 interfaces. **Burr-Brown; $3495.**

Reader Service Number 28

Gone diskless

The IWS 286 DN industrial workstation features a CAT 286 processor and CAT C-RAM memory cards.

The IWS 286 DN factory-hardened, AT-compatible industrial workstation provides a diskless configuration for harsh shop-floor environments. Users execute applications from the RAM/EPROM-based disk systems without need for floppy- or hard-disk drives.

At power-up the system loads MS/PC-DOS operating systems from EPROM. Users can operate the system manually or through an optional batch file loaded from a disk ROM, disk RAM, local area network load, or applications. A 256K-byte disk RAM for user files extends to 15M bytes. **Comark Corp.; $6880.**

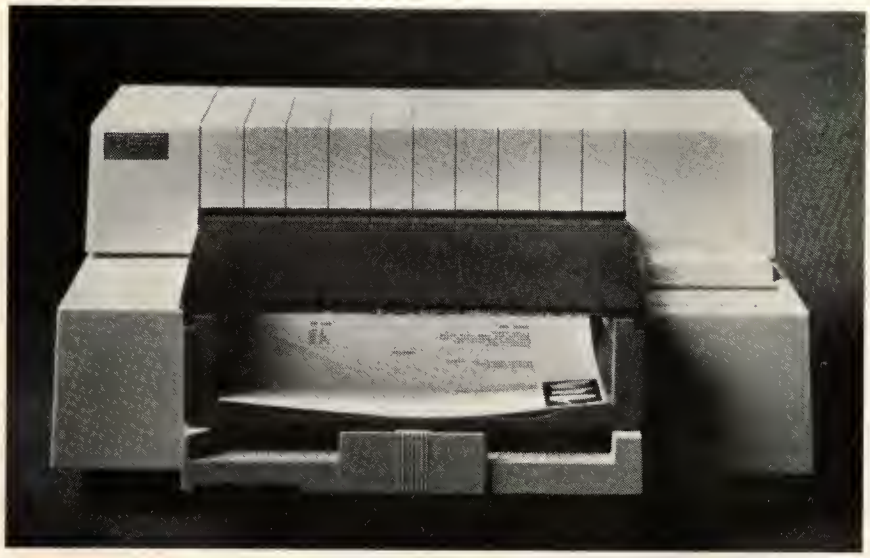
Reader Service Number 29

Read and write all you want

According to the company, RAM-based E100 EPROM Emulator's read/write cycles permit data change without data loss. Compatible with Stag MOS programmers only, this module plugs into micro-based systems to replace the ROM. The E100 emulates 16K- to 512K-bit EPROMs with programming and verification taking 10 seconds. Users can configure multiple units in 8-, 16-, or 32-bit modes. The E100 debugs systems by modification and substitution of existing software. **Stag Microsystems; $650.**

Reader Service Number 30

Lasers for less

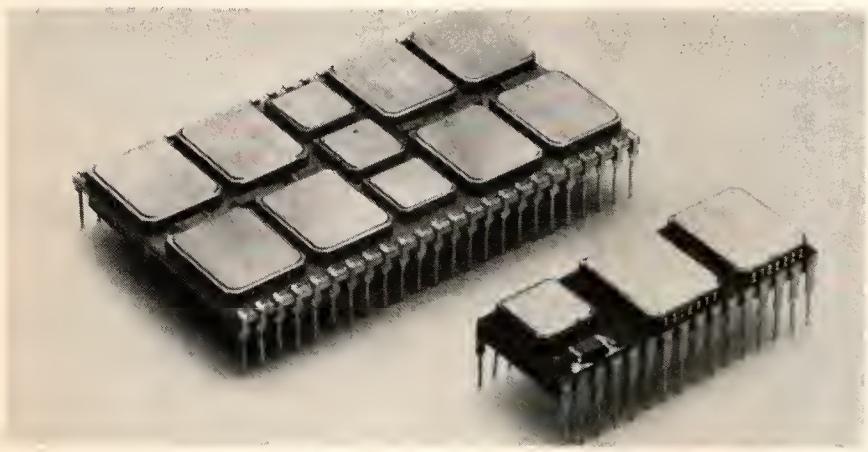


HP's front-loading DeskJet prints laser-quality text at two pages per minute and features an automatic cut-sheet feeder.

Hewlett-Packard offers a laser-quality printer for less than $1000. The IBM-compatible DeskJet printer produces multiple-font text and full-page graphics at 300 dpi. The DeskJet outputs on common office paper and handles merged text and graphics from numerous application packages. **Hewlett-Packard; $995.**

Reader Service Number 31

Military memory modules



EEPROM modules DPE45128 and DPE51288 support defense and aerospace systems requiring nonvolatile memory and minimum board space and power.

Both 1.25-μm DPE45128 and DPE51288 CMOS modules provide fast power-up, false-write protection, an extended page load, and up to 90-ns access times. The 512K × 8-bit DPE45128 contains sixteen 32K × 8-bit EEPROMs in LCC packages, two decoders, three buffers, and a data transceiver on one 48-pin, Cerdip substrate. The 128K × 8-bit DPE51288, featuring a 1M-bit, JEDEC standard pinout, uses four 32K × 8-bit EEPROM devices and a decoder on a 32-pin, cofired ceramic substrate. **Dense-Pac Microsystems.**

DPE45128 **Reader Service Number 32**
DPE51288 **Reader Service Number 33**

In one fell swoop

According to the company, the single-pass CES 30 electrostatic system plots colors simultaneously at 0.3 ips on A- through EE-sized, 254-dpi plots. A built-in raster engine clips, bands, or sorts graphic information concurrently with plotting. The CES 30, which features 2M bytes of RAM and special-purpose VLSI devices, stores complete plot files. An interactive intelligent control panel sets the number of copies, controls color intensities, and performs self-testing procedures. **Benson.**

Reader Service Number 34

Specialty memory used for debugging

The 1DT71502 4K × 16-bit, CMOS static RAM contains 35-ns registers at RAM outputs and a Serial Protocol Channel serial-shift register system. System designers can use this SRAM as a memory device or a debug and diagnostic tool; eight 1DT71502's replace 51 chips. Synchronous/asynchronous output enable permits depth expansion and bus driving. Three chip selects (two programmable) allow cascading of up to eight parts without external decode logic. **Integrated Device Technology; $44 each (100's).**

Reader Service Number 35

Controller operates in extreme temperatures

Designed for automatically activating hydraulic and pneumatic equipment in a 24-volt environment, the 7-in. × 9-in. SBC-64 has a built-in power supply and floating-point Basic for programming I/O and timing. An 8K-byte RAM, self-contained, single-board microcontroller directly drives 32 solenoids and senses 32 opto-isolated AC/DC inputs. Users can expand I/Os and program output voltages to vary beyond 40VDC. Operating temperatures can vary from -20°C to +85°C. **Basicon, Inc.**

Reader Service Number 36

Refresh your memory

When integrating text and graphics, the Quad Pixel Dataflow Manager graphics processor manages and updates bitmapped display memory. The intelligent Am95C60 combines video refresh of the display; memory refresh; update of the bitmap; and arbitration between memory update, video refresh, and dynamic memory refresh. The CMOS QPDM operates at a maximum system clock speed of 20 MHz and allows vectors to be drawn at a rate of up to 3.3 million pps. Bitblt operations replace alphanumeric text controllers.

Built to interface with four memory planes consisting of dual-ported video RAMs, QPDM supports 4K × 4K-bit planes and 2K × 2K-pixel displays. Features include direct text placement into display memory and support for drawing antialiased vectors, circles, and arcs. The graphics processor interfaces to most 8- or 16-bit system buses. **Advanced Micro Devices.**

Reader Service Number 37

Optical system equals 1100 floppies

The MMi-100 optical archiving system stores up to 200M bytes of data on a 5¼-in. optical disk drive. Its OptiDriver software allows users to run most application programs directly from the disk, use standard PC-DOS utilities, and employ same-system multiple device drivers. The optical disk appears as a standard Winchester disk drive, transparent to the operating system. **Optisys; $1495.**

Reader Service Number 38

From history to the printed page

The Codem Disk Reader converts text and format codes from one system to another.

The stand-alone Codem Disk Reader converts 8-in. disks from Wang, Xerox 860, NBI, CPT, and IBM OS6, 5520, and Displaywriter systems to 5.25- or 3.5-in. MS-DOS versions. Disk Reader consists of an 8-in. disk drive, a controller board, and FormScan menu-driven software. Users can install components on IBM PC, AT, or XT systems. Ventura, Aldus, or Xerox 6085 Documenter electronic publishing systems import documents. **FormScan; $7500 with one set of conversion modules.**

Reader Service Number 39

Smaller than a mouse

The 4.25-in.-wide PC-TRAC trackball for MS/PC-DOS computers comes in both serial and bus-interfaced versions.

Based on the FastTRAP pointing device, the PC-TRAC trackball stationary input device features ballistic gain with user-selectable gain tables. Users adjust pointing device sensitivity to match applications. Standard hardware features include 200-ppi resolution, three input buttons, and emulation of Microsoft's serial mouse. The 4.25-in. PC-TRAC is compatible with Lotus 1-2-3, SuperCalc, Wordstar, WordPerfect, and dBaseIII. **Microspeed; $139 (bus version); $119 (serial version).**

Bus version
Reader Service Number 40

Serial version
Reader Service Number 41

Micro image bank captures video

With PicturePower 1.7 software and PS/2 on-board color graphics, users can place video camera images in databases along with text and then transmit picture databases between microcomputers. A 256-color capture board digitizes images of people, products, or signatures. PicturePower also supports MS-DOS computers, which require a capture/display board and an analog monitor in addition to the video camera. Users output pictures and text to film recorders, laser printers, and video printers. The company offers a free image capture board to PS/2 users until September 1. **PictureWare; $995.**

Reader Service Number 42

A pair of flatbeds

The Design Plus and Pro-Scan Hi-Res scanners produce photo burn-ins, detailed retouches, and comprehensives for print-related design work on Genigraphics 100D-Plus and PGP workstations. Both flatbed scanners offer 300-dpi resolution and produce images from reflective and transparent media containing three times the digital information of a video resolution image.

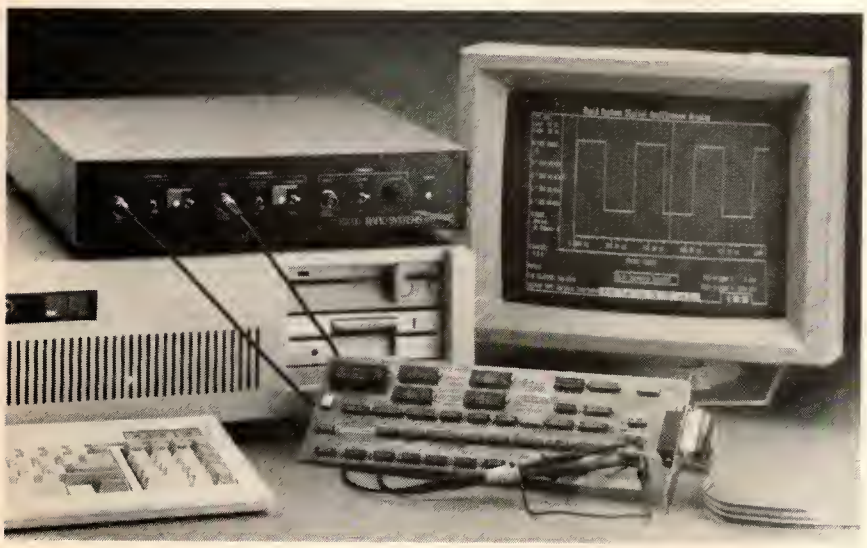
The Design Plus scanner uses the 100D-Plus Paint program to retouch images while maintaining system responsiveness. Users store images on line, merge them with hard-edged vector text, graphs, and illustrations, and display them on screen.

Users retouch Pro-Scan Hi-Res scanner images with PGP Paint 24 and merge graphics and vector illustration with the chART software program. **Genigraphics; $29,000 (Design Plus, available July 1); $19,900 (Pro-Scan Hi-Res, available Aug. 1).**

Design Plus
Reader Service Number 43

Pro-Scan Hi-Res
Reader Service Number 44

Turn your PC into a recorder

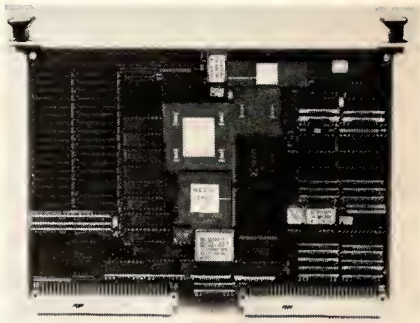


The R2040 recorder features assembly-language drivers, two 8-bit A/D converters for simultaneous input, and expandable 64K-byte data memory per channel.

The R2040 transient recorder peripheral for MS-DOS computers provides a two-channel, 20-MHz simultaneous sampling system. Each channel possesses 64K-byte data memory that is expandable to 128K bytes. Example data acquisition and data display programs complete with source code accompany the unit. Users can program in Basic, Turbo Pascal, and C. **Rapid Systems; $3495.**

Reader Service Number 45

Heurikon starts new family



The HK32/V532 features a VMEbus interface architecture with 32-bit data path, 32-bit addressing, and seven bus interrupts in either master or slave mode. System-level controller functions include single-level arbitration.

Designed around the 32-bit NS32532, the HK32/V532 single-board microcomputer operates in Unix or real-time applications as well as development environments.

An orthogonal CPU architecture with symmetrical instruction set for HLL support, 4G-byte virtual and physical address space, and 370,000 transistor sites offers up to 30-MHz performance levels.

On-chip elements include a 1024-byte, two-way set associative data cache; a 512-byte, direct-mapped instruction cache; and a memory management unit with a 64-entry, fully associative translation lookaside buffer.

Other memory features include a four-channel DMA controller supporting up to 4M-byte/s SCSI transfers; 4M bytes or 16M bytes of on-board, dual-access DRAM with parity; a 32-pin ROM socket for up to 1M byte of EPROM; and 128 bytes of nonvolatile RAM in a 256K × 4-bit configuration. **Heurikon.**

Reader Service Number 46

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 180 **Medium** 181 **High** 182

Advertiser/Product Index

FOR DISPLAY ADVERTISING INFORMATION CONTACT:

Northern California and Pacific Northwest: Don Farris Co., 161 W. 25th Ave., Ste. 102B, San Mateo, CA 94403; (415) 349-2222.
Jack Vance, P.O. Box 3205, Saratoga, CA 95070.
Southern California and Mountain States: Richard C. Faust Co., 24050 Madison St., Ste. 100, Torrance, CA 90505; (213) 373-9604.
Southwest: The House Co., 3000 Weslayan, Ste. 345, Houston, TX 77027; (713) 622-2868.
Midwest: Robert Acker & Associates, 480 Central Ave., Northfield, IL 60093; (312) 441-6050.
East Coast: Atlantic Representative Group, 349 Maple Place, Keyport, NJ 07735; (201) 739-1444.
New England: Arpin Associates, PO Box 227, Weston, MA 02193; (617) 899-5613.
Europe: Heinz J. Gorgens, Parkstasse 8a, D-4054 Nettetal 1 - Hinsbeck (F.R.G.); (0 21 53) 8 99 88.

Advertising Director: Dawn Peck.
For production information, conference, and classified advertising, contact Heidi Rex or Marian Tibayan.

IEEE Micro, 10662 Los Vaqueros Cir., Los Alamitos, CA 90720; (714) 821-8380.

	RS #	Page#
BOARDS		
Controller	36	101
Graphics processor	37	102
Transaction processor	28	100
CHIPS		
EEPROM module	32-33	101
DATA ACQUISITION		
Amplifier	27	100
Emulator	30	100
Transient recorder	45	103
DATA COMMUNICATIONS		
Disk reader	39	102
Drive array	23	99
Pocket modem	2	90
I/O RELATED EQUIPMENT		
Desktop facsimile transceiver	16-17	98
Desktop scanner	24-26	99
Flatbed scanner	43-44	103
Laser-quality printer	31	101
Pointing device	40-41	102
Portable facsimile machine	14-15	98
Thermal transfer printer	18-20	98
MEMORY/STORAGE EQUIPMENT		
CMOS static RAM	35	101
Micro image bank	42	103
Optical archiving system	38	102
PUBLISHERS		
Parallel connection	5	27
RECRUITMENT		
Classified advertising	—	104
SOFTWARE		
Drawing program	3	91
Flat pattern development	11	97
Simulation package	—	Cover IV
SYSTEMS		
Aural computing	22	99
Color video controller	21	98
Development system	12-13	98
Electrostatic system	34	101
Field computer	10	97
Industrial workstation	29	100
Information management	4	92
Microcomputer	46	103
Productivity tool	1	1

Classified Ads

RATES: $5.00 per line, $50 minimum charge (up to ten lines). Average five typeset words per line, eight lines per column inch. Add $4 for box number. Send copy at least one month prior to publication to: **Heidi Rex or Marian Tibayan, Classified Advertising, IEEE MICRO Magazine, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720.**

READER SERVICE CARD

IEEE MICRO

June 1988 issue
(card void after October 1988)

Name_____
Title_____
Company_____
Address_____
City_____ State___ ZIP_____
Country_____ Phone (____) _____

Please send
(Circle those you want):

201 Publications catalog
202 Membership information
203 Student membership information
204 *IEEE Micro* subscription information

Reader interest
(Circle what you liked, add comments on the back)

Readers,
Indicate your interest in articles and departments **by circling the appropriate number** (shown on the last page of articles/departments) **in the shaded section of this card under Product Inquiries.**

Product inquiries **1**
(circle the numbers for products and advertisers you want more information on)

1	21	41	61	81	101	121	141	161	181
2	22	42	62	82	102	122	142	162	182
3	23	43	63	83	103	123	143	163	183
4	24	44	64	84	104	124	144	164	184
5	25	45	65	85	105	125	145	165	185
6	26	46	66	86	106	126	146	166	186
7	27	47	67	87	107	127	147	167	187
8	28	48	68	88	108	128	148	168	188
9	29	49	69	89	109	129	149	169	189
10	30	50	70	90	110	130	150	170	190
11	31	51	71	91	111	131	151	171	191
12	32	52	72	92	112	132	152	172	192
13	33	53	73	93	113	133	153	173	193
14	34	54	74	94	114	134	154	174	194
15	35	55	75	95	115	135	155	175	195
16	36	56	76	96	116	136	156	176	196
17	37	57	77	97	117	137	157	177	197
18	38	58	78	98	118	138	158	178	198
19	39	59	79	99	119	139	159	179	199
20	40	60	80	100	120	140	160	180	200

READER SERVICE CARD

IEEE MICRO

June 1988 issue
(card void after October 1988)

Name_____
Title_____
Company_____
Address_____
City_____ State___ ZIP_____
Country_____ Phone (____) _____

Please send
(Circle those you want):

201 Publications catalog
202 Membership information
203 Student membership information
204 *IEEE Micro* subscription information

Reader interest
(Circle what you liked, add comments on the back)

Readers,
Indicate your interest in articles and departments **by circling the appropriate number** (shown on the last page of articles/departments) **in the shaded section of this card under Product Inquiries.**

Product inquiries **2**
(circle the numbers for products and advertisers you want more information on)

1	21	41	61	81	101	121	141	161	181
2	22	42	62	82	102	122	142	162	182
3	23	43	63	83	103	123	143	163	183
4	24	44	64	84	104	124	144	164	184
5	25	45	65	85	105	125	145	165	185
6	26	46	66	86	106	126	146	166	186
7	27	47	67	87	107	127	147	167	187
8	28	48	68	88	108	128	148	168	188
9	29	49	69	89	109	129	149	169	189
10	30	50	70	90	110	130	150	170	190
11	31	51	71	91	111	131	151	171	191
12	32	52	72	92	112	132	152	172	192
13	33	53	73	93	113	133	153	173	193
14	34	54	74	94	114	134	154	174	194
15	35	55	75	95	115	135	155	175	195
16	36	56	76	96	116	136	156	176	196
17	37	57	77	97	117	137	157	177	197
18	38	58	78	98	118	138	158	178	198
19	39	59	79	99	119	139	159	179	199
20	40	60	80	100	120	140	160	180	200

SUBSCRIPTION CARD

SUBSCRIBE TO IEEE MICRO

☐ YES, sign me up! ☐ Renew my subscription!

If you are a member of the Computer Society or any other IEEE society, pay the member rate of only **$18** for a year's subscription (6 issues). Subscriptions are annualized with membership. For orders submitted March through August, please pay half the given full-year fee for a half-year's subscription.

Society:_____ IEEE membership no:_____

☐ YES, sign me up! ☐ Renew my subscription!

If you are a member of ACM, NSPE, ACS, IEE (UK), SCS, IPSJ, IECEJ, or any other technical society, pay the sister society rate of only $27 for a year's subscription (6 issues).

Society:_____ Mem. no. (if any):_____

_____
Full Signature *Date*

_____
Name

_____
Street

_____
City

_____
State/Country *ZIP/Postal Code*

☐ Payment enclosed
☐ Charge to ☐ Visa ☐ MasterCard ☐ AmEx

☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐
Charge Card Number

Mo. Yr.
☐☐ ☐☐
Exp. Date

M688

Charge orders also taken by phone:
(714) 821-8380 8:00 a.m. to 5:00 p.m. Pacific time
Circulation Dept.
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720

Editorial comments

I liked: _____

I disliked: _____

I would like to see: _____

For reader service inquiries, use other side

PO box address for reader service cards only

PLACE STAMP HERE

IEEE MICRO

Reader Service Inquiries
PO Box 16508
North Hollywood, CA 91615-6508
USA

Editorial comments

I liked: _____

I disliked: _____

I would like to see: _____

For reader service inquiries, use other side

PO box address for reader service cards only

PLACE STAMP HERE

IEEE MICRO

Reader Service Inquiries
PO Box 16508
North Hollywood, CA 91615-6508
USA

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 38 LOS ALAMITOS, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Computer Society of the IEEE
Circulation Dept.
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720-9970

THE COMPUTER SOCIETY

A member society of the Institute of Electrical and Electronics Engineers, Inc.

Computer Society Offices

Headquarters Office
1730 Massachusetts Ave. NW
Washington, DC 20036-1903
Phone: (202) 371-0101
Telex: 7108250437 IEEE COMPSO

Publications Office
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
Membership and General Information: (714) 821-8380
Publications Orders: (800) 272-6657

European Office
13, Avenue de l'Aquilon
B-1200 Brussels, Belgium
Phone: 32 (2) 770-21-98
Telex: 25387 AVVALB

Use the Reader Service Card to obtain the following information:

- Membership application—student #203, others #202
- Periodicals subscription form for individuals #200
- Periodicals subscription form for organizations #199
- Publications catalog #201
- Standards working groups list #195
- COMPMAIL+ international electronic mail/database brochure #194
- Technical committee list/application #197
- Chapters lists, start-up procedures—student/regular #193
- Student scholarship information #192
- Awards description/nomination forms #198
- Volunteer leaders/staff directory #196
- IEEE senior member application #204

Purpose

The Computer Society advances the theory and practice of computer science and engineering, promotes the exchange of technical information among 90,000 members worldwide, and provides a wide range of services to members and nonmembers.

Membership

Members receive the acclaimed monthly magazine *Computer,* discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others seriously interested in the computer field.

Publications and Activities

Computer. An authoritative, easy-to-read magazine containing tutorial and in-depth articles on topics across the computer field; plus news, conferences, calendar, interviews, new products, etc.

Periodicals. The society publishes six magazines and three research transactions. Refer to membership application or request information as noted above.

Conference Proceedings, Tutorial Texts, Standards Documents. The Computer Society Press publishes more than 100 titles every year.

Standards Working Groups. Over 100 of these groups produce IEEE standards used throughout the industrial world.

Technical Committees. Over 30 TCs provide newsletters, interaction with peers in specialty areas, and have direct influence on standards, conferences, education, etc.

Conferences/Education. The society holds about 100 conferences each year and sponsors many educational activities, including computing sciences accreditation.

Chapters. Regular and student chapters worldwide provide the opportunity to interact with colleagues, hear technical experts, and serve the local professional community.

European Office

Payments for Computer Society membership and publication orders are accepted by cheques in Belgian, British, German, Swiss, or U.S. currency, or by American Express, Eurocard, MasterCard, or Visa credit cards.

Ombudsman

Members experiencing problems—magazine delivery, membership status, unresolved complaints—may write to the ombudsman at the Publications Office.

NEW FOR NETWORK DESIGNERS

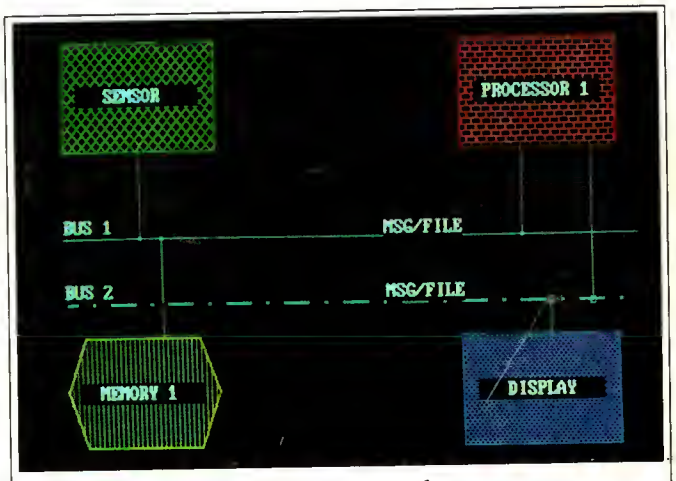
BEFORE

```
        /* PLACE ENTITY IN WAITING LINE */
        IF NQUE=0 THEN FIRST=P;
                  ELSE LAST->FOLL=P;
        CUMQ=CUMQ+NQUE*(CLOCK-TLAST);
        TLAST=CLOCK;
        NQUE=NQUE+1;
        NMAX=MAX(NMAX,NQUE);
        LAST=P;
        TNXT_ARV=CLOCK+RANDRATE_FROM,RATE_TO);
        RETURN;
        END;

        /* DEPARTURE ROUTINE */
DEPART: PROCEDURE;
        /* FREE BOOTH */
        CTEL=CTEL+TEL->USE_TIME;
        STATE=0;
        FREE TEL->USER;
        IF NQUE=0 THEN DO;
             /* SYSTEM IS EMPTY. FORCE NEXT */
             /* EVENT TO BE ARRIVAL.        */
             TNXT_DPRT=1E50;
             RETURN;
             END;
        CUMQ=CUMQ+NQUE*(CLOCK-TLAST);
        TLAST=CLOCK;
        CUMT=CUMT+      (CLOCK-FIRST->ARVL_TIME);
        /* PLACE FIRST WAITING CALLER IN BOOTH */
        TEL=FIRST;
        FIRST=TEL->FOLL;
        NQUE=NQUE-1;
        NUSE=NUSE+1;
        STATE=1;
        TNXT_DPRT=CLOCK+TEL->USE_TIME;
        RETURN;
        END;
   /* RANDOM NUMBER GENERATOR */
RAND:   PROCEDURE(FROM,TO);
        DCL (FROM,TO) FIXED BIN;
        DCL (IX,IY) STATIC FIXED BIN(31) INIT(1);
             CALL RANDU(IX,IY,YFL);
             IX=IY;
             RFROM+YFL*(TO+1-FROM);
             RETURN(M);
             END;
```

Costs and delays of programming

AFTER



Computer—communications network simulation--quick results, no programming

See your proposed network perform under various workloads

NETWORK II.5 now gives you easy-to-understand simulation results quickly--no programming

With NETWORK II.5 you enter your computer—communications network description.

Simulation follows immediately --no programming delays.

Easy-to-understand results

Your reports show utilization, queues, execution times, response times and conflicts.

Graphical reports show hardware layout, software data flow, and device utilization.

Seeing graphical results increases everyone's understanding of your proposed network and builds confidence in the analysis.

Your system simulated

You can analyze any computer—communications system including local area networks.

You can simulate some portions of the system at a detailed level and others at a coarser level.

Widely used protocols are built-in --you just make a choice.

Free trial and training offer

The free trial contains everything you need to try NETWORK II.5® on your computer.

You may develop your own network or modify one of ours.

Try the NETWORK II.5 approach, the quality and timeliness of our support, the accuracy of our documentation and the facilities for error-checking--**everything you need for a successful project.**

No cost, no obligation.

Act now-free training offer

For a limited time we also include **free training.** Typical applications are demonstrated.

Call today to avoid disappointment--class size is limited.

For immediate information

Call Paul Gorman at (619) 457-9681. In the UK, call Steve Wombell on (01) 940-3606.

The quickest, easiest way for you to evaluate network alternatives is with NETWORK II.5.

With NETWORK II.5 you get results sooner and they are better understood.

Rush information on the NETWORK II.5 free trial and training offer

Free trial-learn the reasons for the broad and growing popularity of NETWORK II.5—**no cost, no obligation.**

Limited offer-return the coupon today and we will include one free course enrollment worth $850.

Name

Organization

Address

City State Zip

Telephone Computer

IEEE MICRO

Return to:
CACI
3344 North Torrey Pines Court
La Jolla, California 92037
Or, better yet, call Paul Gorman at (619) 457-9681

In the UK
CACI
26 The Quadrant
Richmond, Surrey, England TW9 1DL
Call Steve Wombell on (01) 940-3606